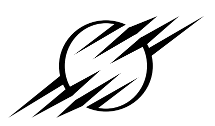
Analyse, Conception et Programmation par Objets

Norbert Kajler et Fabien Moutarde
{Norbert.Kajler, Fabien.Moutarde}@mines-paristech.fr

Année 2011-2012
(Dernière mise à jour le 15 septembre 2011)



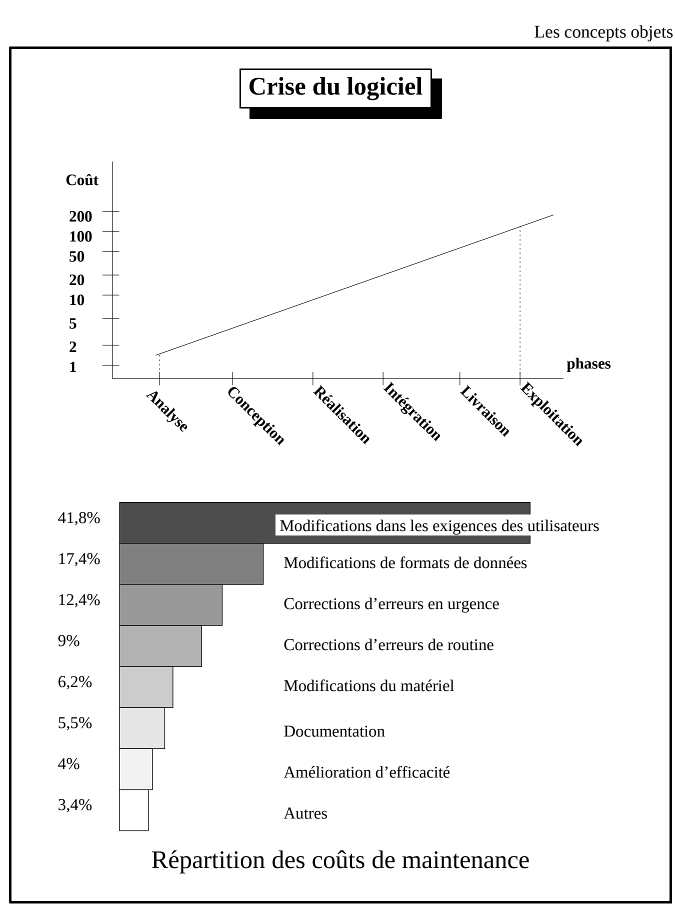
Mines ParisTech
60 Bd Saint Michel
75272 Paris Cedex 06

Plan

Les concepts objets

1. **Pourquoi la programmation par objets.**
2. **Qualité du logiciel.**
3. **Modularité.**
4. **Abstraction et encapsulation.**
5. **Objet, message, méthode.**
6. **Classe et instance.**
7. **Héritage.**
8. **Polymorphisme.**

Crise du logiciel



Répartition des coûts de maintenance

Défis des langages et méthodes objets

Quatre défis principaux :

- ▷ Réduire les coûts de développement des logiciels.
- ▷ Réduire les coûts de maintenance.
- ▷ Faciliter la réutilisation de composants logiciels déjà réalisés.
- ▷ Accroître la qualité du logiciel.

... et une révolution en cours pour les Informaticiens :

- ▷ Développement de bibliothèques de composants logiciels *réutilisables*.
- ▷ Construction d'applications par *assemblage* de composants selon des méthodes rigoureuses et éprouvées.
- ▷ Travail à un plus haut niveau d'abstraction que le code.

Qualité du logiciel

Facteurs externes *(concernent utilisateurs et développeurs)* :

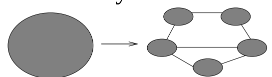
- ▷ **Validité** : aptitude à satisfaire exactement les tâches définies par sa spécification.
- ▷ **Robustesse** : capacité à fonctionner même dans des conditions anormales.
- ▷ **Extensibilité** : facilité d'adaptation d'un logiciel aux changements de spécification.
- ▷ **Compatibilité** : aptitude à pouvoir être combiné à d'autres logiciels.
- ▷ **Efficacité**, **portabilité**, **facilité d'utilisation**, etc.

Facteurs internes *(concernent uniquement les développeurs)* :

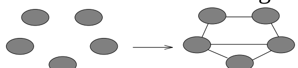
- ▷ **Ré-utilisabilité** : possibilité de réutiliser des éléments d'un logiciel pour réaliser de nouvelles applications.
- ▷ **Vérifiabilité** : facilité de préparation/réalisation des procédures de certification.
- ▷ **Modularité**, **lisibilité**, etc.

Modularité

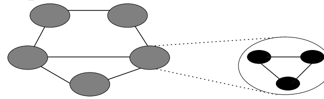
- ▷ **Décomposabilité modulaire** : décomposition d'un logiciel complexe en sous-systèmes moins complexes.



- ▷ **Composabilité modulaire** : utilisation d'éléments existants pour construire de nouveaux logiciels.



- ▷ **Compréhensibilité modulaire** : compréhension de chaque module pris isolément.



- ▷ **Protection modulaire** : chaque module est responsable de ses dysfonctionnements et les traite localement.

Processus d'abstraction

Processus de structuration/simplification de la réalité :

- ▷ Définition des frontières conceptuelles.
- ▷ Suppression des détails non significatifs pour l'application.
- ▷ Concentration sur la vue externe de l'objet.

Exemple : abstraction du concept d'usager dans une bibliothèque :

- ▷ **Attributs :** numéro, nom, prénom, adresse.
- ▷ **Comportement :** emprunter(), rendre(), démissionner().

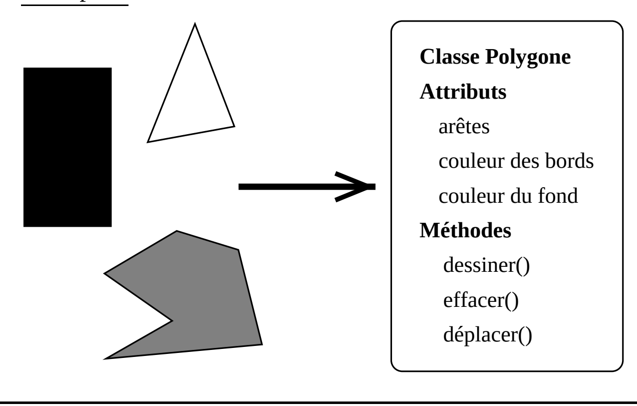
Encapsulation

Technique consistant à regrouper en une même entité (un **objet**) des données et les procédures pour manipuler ces objets.

▷ Les données (ou **attributs**) sont appelées la partie **statique** de l'objet ;

▷ les procédures (ou **méthodes**) sont appelées la partie **dynamique**.

Exemple :

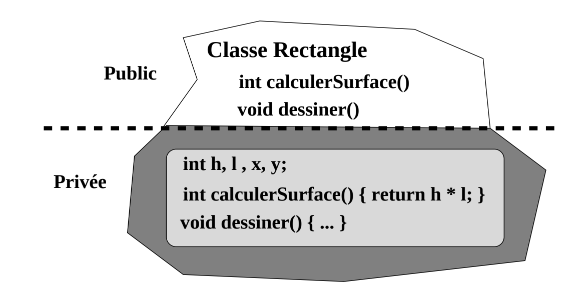


Interface et implantation

Séparation stricte entre :

▷ l'**implantation** de l'objet (ou **partie privée**) ;

▷ et son **interface** (ou **partie publique**).



⇝ L'interface doit être aussi **simple** et **sûre** que possible.

⇝ La partie privée est réservée au concepteur/développeur de l'objet (invisible aux utilisateurs de la classe).

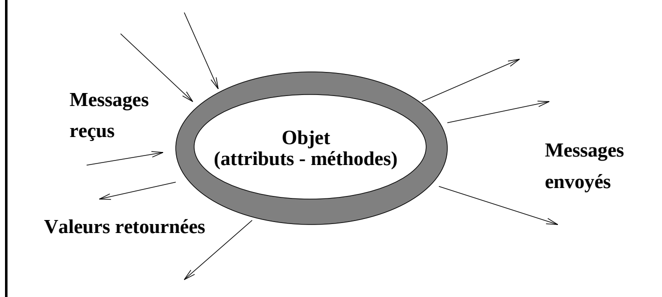
Abstraction de données ≡ partie publique constituée exclusivement de méthodes (pas de données).

Abstraction de données : exercice

Pourquoi l'abstraction de données est une « bonne pratique » ?

Objet, Message, Méthode

Les objets sont des entités autonomes communicantes qui s'échangent des **messages** :



▷ **Message** : requête d'une action que doit réaliser l'objet récepteur.

▷ Les messages sont mis en oeuvre par les **méthodes** de l'objet récepteur.

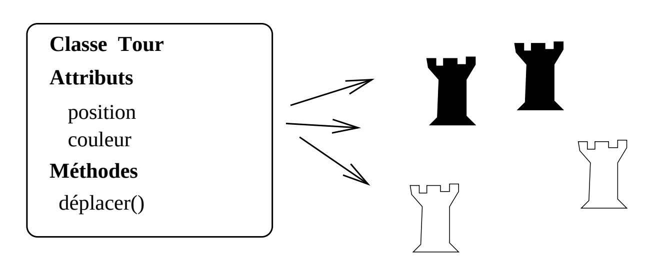
▷ La dénomination message/objet met en évidence l'idée que l'objet est responsable de l'exécution de la requête portée par le message.

Classe et instance

▷ **Classe** : modèle d'objets ayant les mêmes types de propriétés et de comportements.

▷ Chaque **instance** d'une classe possède ses propres valeurs pour chaque attribut.

Exemple :



⇝ Analogie avec le concept type / variable.

Héritage

▷ **Relation** entre deux classes A et B permettant de définir la classe B à partir de la classe A.

▷ La classe mère transmet ses attributs et ses méthodes à ses classes filles.

▷ La classe **fille** ou **sous-classe** est un cas particulier de la ou des classe(s) **mère(s)** ou **super-classe(s)**.

▷ Selon les langages objets :
- héritage simple ou multiple ;
- une ou plusieurs classe(s) racine(s) ;
- plus ou moins grand nombre de classes pré-définies.

Héritage simple : exemple

◇ **Vision hiérarchique :**

Classe Véhicule
Attributs
nb_passager
Méthodes
avancer()

Voiture
Attributs
puissance

2roues

Motocyclette

Bicyclette

relation d'héritage

◇ **Vision ensembliste :**

Véhicule
2roues
Motocyclette
Bicyclette
Voiture

Relation client *vs* Relation d'héritage

▷ **Relation client** : « *a un* »

⇝ relation de composition, de collaboration entre les objets.

Exemples :
- une voiture *a* 4 roues ;
- une voiture *a* 1 moteur.

▷ **Relation d'héritage** : « *est un* »

⇝ relation de spécialisation, relation hiérarchique entre les objets.

Exemples :
- une voiture *est* un véhicule ;
- une Twingo *est* une voiture .

Polymorphisme

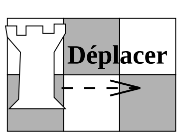
▷ Propriété d'une méthode de pouvoir se comporter de manière différente sur des objets de classes différentes.

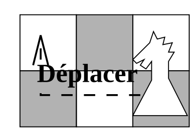
▷ Polymorphisme *statique* :

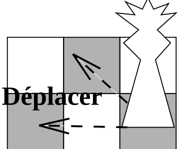
```
double mult(double, double);
double mult(int, double);
int mult(int, int);
```

▷ Polymorphisme *dynamique* :

```
@Piece p;
  ...
p=new Tour(); // ou new Roi() ou ...
  ...
p.deplacer();
```







Avantage du polymorphisme

▷ En programmation structurée (Pascal, C, etc.) :

```
pour chaque pièce faire
   selon type de la pièce
      cas Tour  : deplacer_Tour()
      cas Roi  : deplacer_Roi()
      cas Fou  : deplacer_Fou()
   fin selon
fin pour
```

▷ En programmation par objets (C++, Java, etc.) :

- ○ Définition d'une classe abstraite `Piece`.
- ○ Définition des sous-classes `Tour`, `Roi`, `Fou`, ... avec pour chacune une méthode `deplacer` spécifique.

```
pour chaque pièce p faire
   p .deplacer()
fin pour
```


Analyse et conception

1. **Cycle de vie d'un logiciel**.
2. **Méthode d'analyse et de conception**.
3. **Atelier de Génie Logiciel**.
4. **Analyse et conception objets**.

Cycle de vie d'un logiciel

D'un problème à sa solution Informatique :

▷ **Expression des besoins, spécifications fonctionnelles** : description détaillée des fonctionnalités souhaitées *du point de vue des futurs utilisateurs*.

▷ **Analyse** : spécification du « **quoi** », c-à-d. description détaillée du système à réaliser.

▷ **Conception** : définit « **comment** » le système sera réalisé, et comment il se décompose en modules.

▷ **Réalisation** : codage de chacun des modules.

▷ **Intégration** : test unitaire de chaque module et intégration des différentes modules dans le système.

▷ **Validation** : vérification de la conformité globale du logiciel produit par rapport à sa spécification.

▷ **Maintenance, évolution, ...**

Analyse et conception par objets

Description du monde réel

Analyse

Modèles d'objet et interactions

Conception

Architecture détaillée du système et des modules

Réalisation

Système réalisé avec un langage à objets

Cycle en cascade

Enchaînement linéaire des phases :

Analyse

Conception

Réalisation

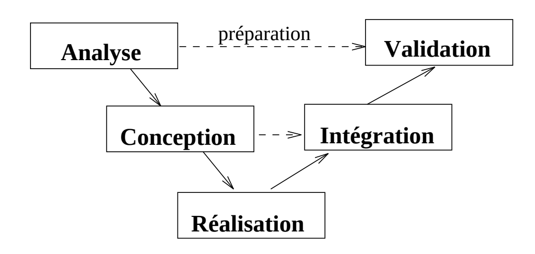
Intégration

Validation

⇝ Cycle très rigide ;
⇝ retour en arrière extrêmement coûteux.

Le cycle en V

Mise en parallèle des phases de construction et de vérification :



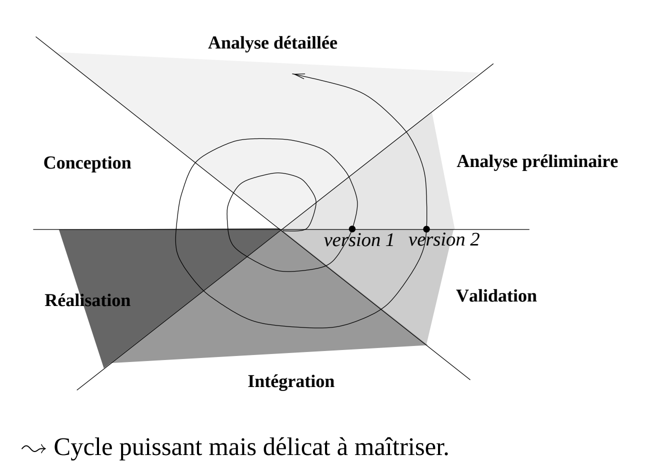
⇝ Relative rigidité ;
⇝ cycle « classique » car relativement facile à maîtriser et à mettre en oeuvre.

Cycle en spirale (ou « itératif »)

▷ Chaque phase du cycle est ponctuée par une analyse des risques encourus sur la phase suivante.

▷ Si les risques sont importants l'équipe aura recourt à du prototypage.

▷ Chaque itération fournit une nouvelle version du système (pas forcément complète).



⇝ Cycle puissant mais délicat à maîtriser.

Méthodes d'analyse et de conception

⇝ « Méthodologie » ou « software development process »

▷ Objectifs :

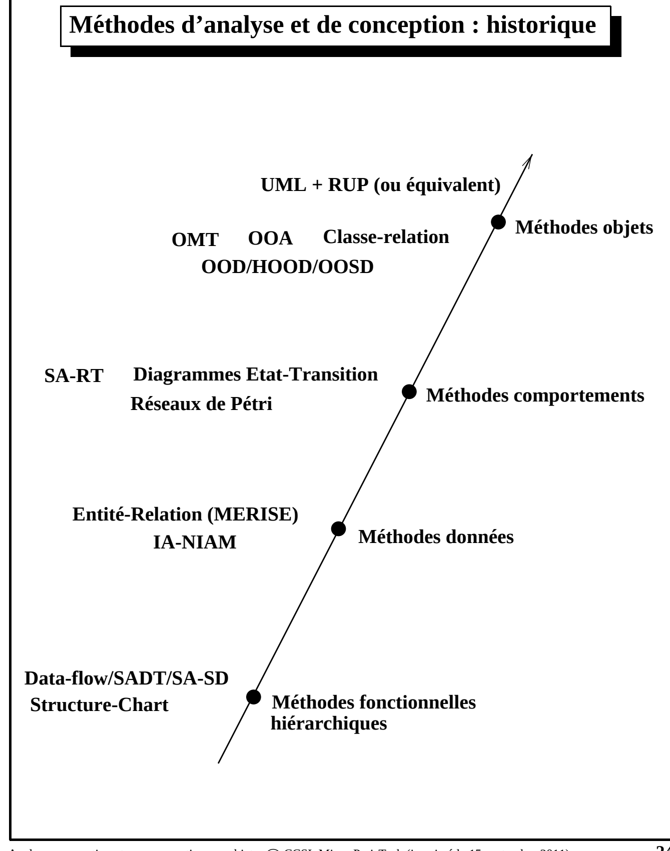
- permettre la spécification du logiciel à l'aide d'un langage de modélisation (e.g. UML) ;
- guider les différents intervenants lors des différentes phases du développement logiciel ;
- conduire à un meilleur contrôle des facteurs économiques pertinents : qualité, coût, délais, risques.

▷ Nécessite en pratique :

- un outil (logiciel) pour saisir/manipuler/vérifier les spécifications exprimées dans le langage de modélisation, ainsi que pour superviser le déroulement du projet.
- une formation préalable des intervenants au projet.

▷ Exemple de méthodologies : OMT, Booch, UP/RUP, ...

Méthodes d'analyse et de conception : historique



Atelier de Génie Logiciel (ou «AGL»)

▷ Un AGL rassemble typiquement :

- des éditeurs graphiques et syntaxiques permettant de saisir/vérifier l'ensemble d'une modélisation ;
- un ou plusieurs générateurs de code ;
- un générateur de documentation ;
- une suite d'outils de Génie Logiciel tels que :
  - gestionnaire de versions ;
  - gestionnaire de tests ;
  - analyseur de performances ;
  - ...

▷ Exemples d'AGL : Rational Software Modeler (Rose), Together, Modelio, Visual Paradigm, Entreprise Architect, ...

▷ Alternatives gratuites : Bouml, StarUML, ...

UML : introduction

1. **Fondements**.
2. **Historique et perspectives**.
3. **Disponibilité**.
4. **Exemple**.
5. **Contenu d'UML**.

Fondements

▷ Principales qualités du modèle objet :

- association données-traitements (encapsulation) ;
- séparation entre interface et implantation (public/privé) ;
- organisation hiérarchique des classes (héritage).

▷ Principales limites du modèle (et des langages) objet :

- modélisation des relations non-hiérarchiques ;
- vision dynamique de l'application ;
- méthodologie (découverte des classes, génération et documentation du code, reverse engineering, etc.)

⇝ **méthodes d'analyse et de conception objets** telles que : OMT, OOAD, Classe-Relation, UML + RUP, UML + Catalysis, etc.

Historique

Unified Modeling Language (UML) :

▷ **Langage de modélisation** «universel»

▷ Résultat de la convergence en 1995 des méthodes :
– OMT (James Rumbaugh),
– OOA-OOD (Grady Booch),
– Objectory/OOSE (Ivar Jacobson)

▷ **Standard industriel** sous la direction de l'Object Management Group (OMG) depuis UML 1.1 (1997) :

▷ Versions utilisées : UML 1.5 ⇝ UML 2.0 (2005)

▷ Extensions (« Profiles ») :
– UML Profile for Enterprise Distributed Object Computing (EDOC) ;
– UML P. for Schedulability, Performance and Time ;
– UML Testing Profile ;
– ...

Disponibilité

▷ Standard disponible librement sur le site Web de l'OMG
⇝ `http://www.omg.org/uml/`
(documentation de référence très technique, plutôt destinée à des utilisateurs avertis : auteurs d'ouvrages, concepteurs d'outils UML notamment).

▷ Supporté par de nombreux AGL (Rational Software Modeler, Modelio, Rhapsody, Together, ...) permettant la modélisation, génération de code (C++, Java, C#, etc.), génération des documentations, retro-conception, ...

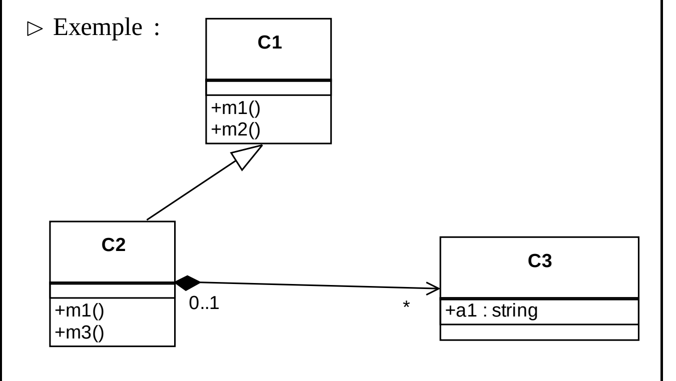
▷ Documentation :
– ces transparents ;
– livres et sites web sur UML (cf. Bibliographie) ;
– `http://www.ensmp.fr/CC/Docs`

Formalisme utilisé dans UML

▷ Essentiellement graphique.

▷ Notations spécifiques pour chacun des concepts UML : packages, classes, relations, instances, états, etc.

▷ Exemple :



▷ Besoin d'un outil adapté : édition des diagrammes, gestion des cohérences entre diagrammes, génération de la doc et du code, etc.
⇝ AGL comme « Rational Software Modeler (Rose) », « Modelio », ...

Contenu d'UML : survol rapide

▷ Une notation unifiée

⇝ facilite la compréhension / communication d'une modélisation.

▷ Un méta-modèle complètement spécifié

⇝ spécifie formellement la sémantique d'UML ;
⇝ facilite la réalisation d'outils et l'échange d'une modélisation entre outils.

▷ Attention : UML n'impose pas de « processus de développement » (ou « méthodologie ») spécifique

⇝ possibilité d'adapter UML aux spécificités de chaque organisation et/ou projet en terme de processus de développement.

Contenu d'UML : hiérarchie des concepts

▷ Les « *diagrammes* » (voir p.33)

▷ Les « *choses* » (ou *things*)

- structurelles :
  ⇝ **classes**, **interfaces**, **collaborations**, **cas d'utilisation**, **classes actives**, **composants**, **noeuds**
- dynamiques (ou *behavioral*) :
  ⇝ **interactions**, **automates**
- groupantes :
  ⇝ **packages**
- annotantes :
  ⇝ **notes**

▷ Les « *relations* » :
⇝ **dépendances**, **associations**, **généralisations**

▷ L'« *extensibilité* » :
⇝ **stéréotypes**, **propriétés**, **contraintes**, **commentaires**

Contenu d'UML : neuf types de diagramme

▷ Aspects « fonctionnels » :

- Diagramme de **cas d'utilisation** (ou **use case** diagram)
  ⇝ [Expression des besoins / An.]

▷ Aspects « structurels » :

- Diagramme de **classes** ⇝ [An. / Co.]
- Diagramme d'**instances** (ou d'**objets**) ⇝ [An.]
- Diagramme de **composants** ⇝ [Co.]
- Diagramme de **déploiement** ⇝ [Co.]

▷ Aspects « dynamiques » :

- Diagramme **d'états-transitions** (ou **statechart** diagram) ⇝ [An. / Co.]
- Diagramme de **séquence** (ou **scénario)** ⇝ [An.]
- Diagramme de **collaboration** ⇝ [An.]
- Diagramme d'**activités** ⇝ [An.]

UML : modèle des classes

1. **Packages**.
2. **Classes**.
3. **Attributs, opérations, invariants et pré-/post-conditions**.
4. **Associations et agrégations**.
5. **Généralisation**.
6. **Instances et diagrammes d'instances**
7. **Règles d'emploi du modèle des classes**.

Packages

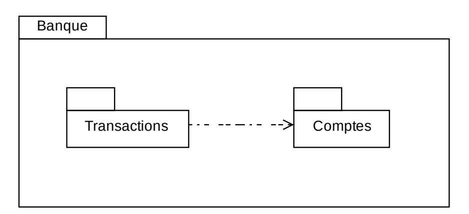
▷ **Encapsulation** des classes en distinguant :

- les classes *privées* (classes strictement internes au package) ;
- les classes *protégées* (classes pouvant être utilisées par des classes appartenant à des packages « fils ») ;
- les classes *publiques* (classes pouvant être utilisées par des classes de n'importe quel package).

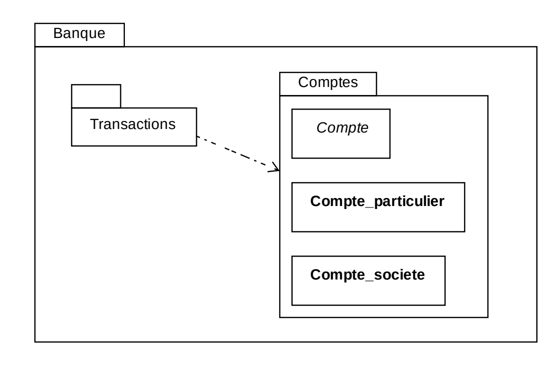
▷ Liens de généralisation, de référencement et de composition entre packages (un package peut encapsuler d'autres packages).

Packages : exemple

◇ **Un package peut comporter d'autres packages :**



◇ **Dans chaque diagramme on contrôle librement le niveau de détail :**



Classes

▷ Une classe appartient à un (et un seul) package.

▷ Rappel : une classe peut être publique, privée ou protégée au sein de son package.

▷ Une classe peut comporter des *attributs*, des *opérations* et un *invariant*.

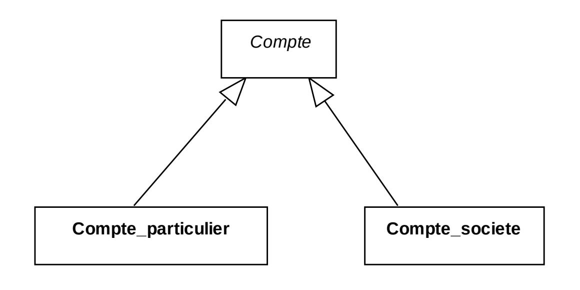
▷ Une classe peut être en relation avec :
– des classes du même package ;
– des classes publiques appartenant à des packages « visibles » depuis le package courant.

▷ Une classe peut être abstraite (au sens Java/C++)
⇝ notation : *nom* de la classe en italique...
...ou alors écriture de la propriété `{abstract}` sous le nom.

Remarque : deux classes peuvent porter le même nom à condition de ne pas appartenir au même package.

Packages et classes : exemple

◇ **Le package Comptes comporte trois classes :**



⇝ la classe Compte est abstraite (italique) ;

⇝ la classe Compte est la classe mère des deux autres classes du package.

Remarque : à l'extérieur de leur package de définition les noms de classes sont préfixés par le nom du package...
⇝ Exemple : `Comptes::Compte_societe`

Attribut

▷ Un attribut possède :

- un nom ;
- un type ;
- éventuellement une valeur par défaut.

▷ Un attribut peut être « de classe » (≡ `static` en C++/Java)...
⇝ notation : <u>nom et type de l'attribut</u> souligné.

▷ Exemple :

Salaire
date_paye : date indice : integer=0 <u>facteur : real</u>

▷ Types élémentaires pré-définis en UML : boolean, char, integer, real, string.

▷ Possibilité de définir de nouveaux types élémentaires au niveau d'une classe ou d'un package.

Invariant de classe

▷ Ensemble de contraintes d'intégrité que doit satisfaire l'ensemble des instances de la classe.

▷ Ces contraintes expriment tout ce qui ne peut pas être exprimé par les attributs et les opérations.

▷ Expression syntaxique uniquement : sous forme textuelle et/ou sous forme d'expressions booléennes dans le langage d'implantation final (e.g. C++ ou Java) ou en langage « OCL » (Object Constraint Language).

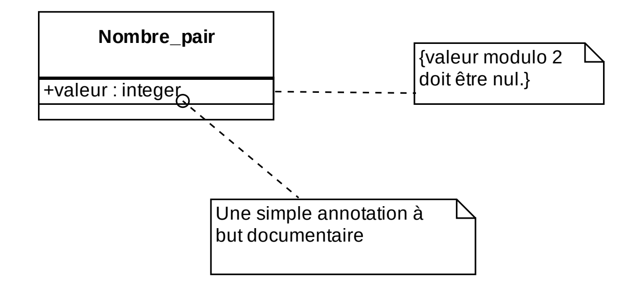
Exemples :

– L'invariant de la classe `NombrePair` est que le modulo 2 de la valeur de l'attribut `value` doit être nul (ou en Java : `value%2 == 0`).

– Soit une classe `Carré` héritière d'une classe `Rectangle`, son invariant peut s'écrire en Java : `largeur == hauteur`

Invariant de classe : exemple

◇ **La contrainte associée à l'attribut valeur a été déclarée comme un invariant :**



<u>Remarque :</u> outre l'invariant, on a associé une note à l'attribut valeur. En fait on peut associer une note à toute entité UML et ce dans n'importe quel diagramme.

Opération

▷ Une opération est définie par :

- son nom ;
- ses paramètres : nom, type, valeur par défaut, mode de passage :
  – passage par valeur : **In** ;
  – passage par référence pour un résultat : **Out** ;
  – passage par référence d'une valeur qui pourra être modifiée : **Inout** ;
- son type de retour éventuel ;
- son action sur l'instance courante :
  – modification possible (mode par défaut) ;
  – opération « constante » : propriété `query`.

▷ Une opération peut être « de classe »
⇝ notation : <u>souligné</u>

▷ Une opération peut être abstraite
⇝ notation : en *italique* ou étiquette `{abstract}`

Classe, attribut, opération : exemple

Salaire
date_paye : date indice : integer=0 facteur : real
augmenter(In montant:euro) calculer_salaire():euro donner_avance(In montant:euro) payer() calculer_moyenne_salaires():euro

Question : il y a plusieurs anomalies dans cette spécification, lesquelles ?

Pré-/Post-conditions

▷ **Pré-conditions** d'une opération : ensemble des conditions préalables à l'exécution de l'opération.

▷ **Post-conditions** d'une opération : ensemble des conditions devant être vérifiées à l'issue de l'exécution de l'opération.

▷ Elles sont toujours sous forme syntaxique :
- *texte* à usage documentaire ;
- *expression* en langage OCL ;
- *expression booléenne* exprimée dans le langage d'implantation (e.g. Java ou C++).

▷ Elles comprennent toujours implicitement l'invariant de la classe. Cas particuliers :
- pour le constructeur, l'invariant n'est pas ajouté à la pré-condition mais il l'est à la post-condition ;
- pour le destructeur, l'invariant est ajouté à la pré-condition, mais ne l'est pas à la post-condition.

Pré-/Post-conditions : exemple

Soit l'opération :

```
débiterCompte(montant : in Euros):
```

Pré-condition : Le solde du compte est supérieur à « montant ».

Post-condition : Le solde du compte a bien été diminué de « montant ».

Visibilité des attributs, opérations et associations

▷ **Public** ... notation : nom préfixé par **+**
⇝ *élément accessible à tous.*

▷ **Package** ... notation : nom préfixé par ~
⇝ *élt. accessible aux seules instances des classes appartenant au même package.*

▷ **Protégé** ... notation : nom préfixé par **#**
⇝ *élt. accessible aux seules instances de la classe, ainsi qu'aux instances des classes dérivées.*

▷ **Privé** ... notation : nom préfixé par **–**
⇝ *élt. accessible seulement aux instances de la classe.*

Exemple :

Salaire
#date_paye : date #indice : integer=0 #facteur : real
+augmenter(In montant:euro) +calculer_salaire():euro +donner_avance(In montant:euro) +payer() +calculer_moyenne_salaires():euro

Association

Avion — véhicule_propulsé — +propulseur — Réacteur

1-1 — *propulsion* — 2-*

▷ Une association *peut* avoir :

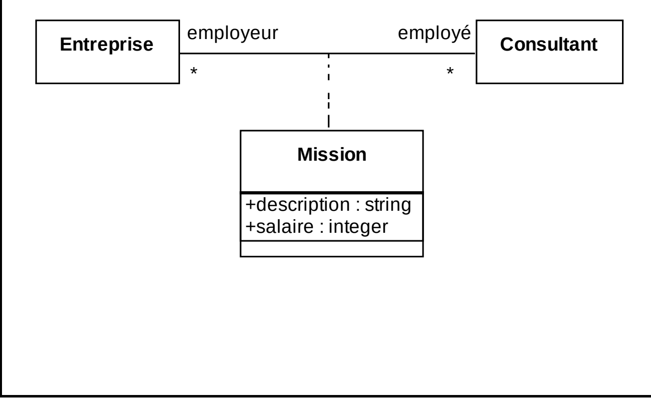
- un nom ;
- deux rôles ;
- une orientation (voir p.49) ;
- deux cardinalités — la *cardinalité* indique les nombres d'instances minimum et maximum pouvant être liées à une instance de la classe.
  Exemples : 2 * 0..2 1..* 1..10, 50, 100

Classe d'association

▷ Permet d'attacher des propriétés (attributs et/ou opérations) directement à l'association liant deux classes.

▷ Attention : une classe d'association ne peut être reliée qu'à une seule association (par contre elle peut être en relation avec d'autres classes).

▷ Exemple :



Association orientée

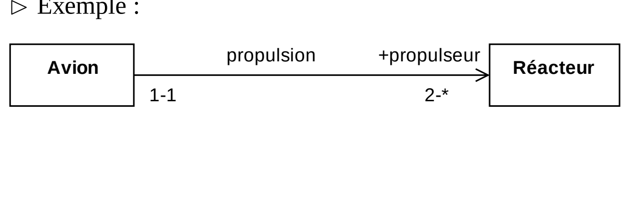
▷ L'orientation spécifie un sens d'utilisation, c-à-d. : la classe qui utilise *vs* la classe qui est utilisée.

▷ Une association peut ainsi :

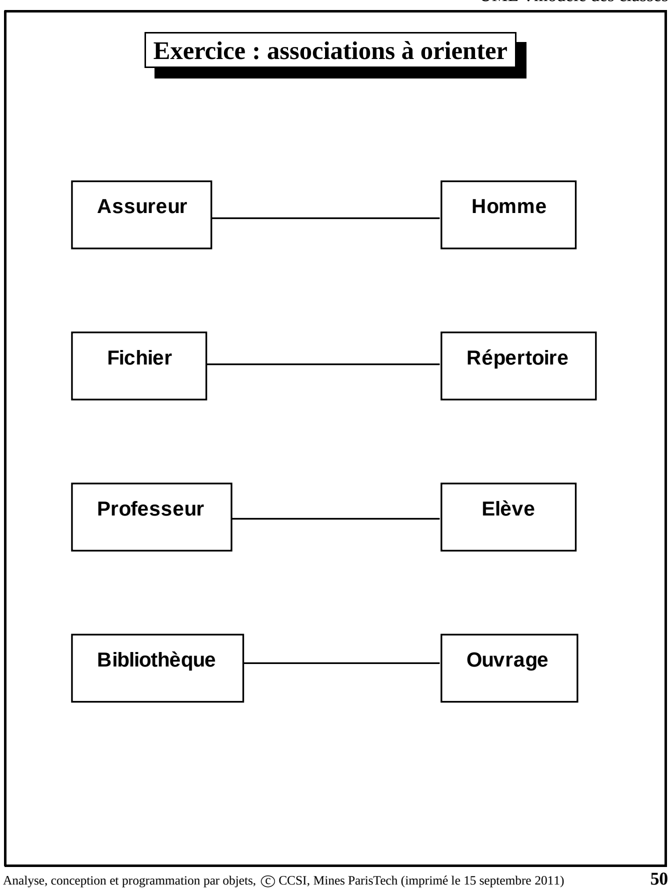
- être orientée dans un sens ;
- être orientée dans les 2 sens ;
- ne pas être orientée.

▷ A l'issu de l'analyse, toutes les associations devraient être orientées, et, sauf cas très particulier, dans un seul sens.

▷ Exemple :



Exercice : associations à orienter

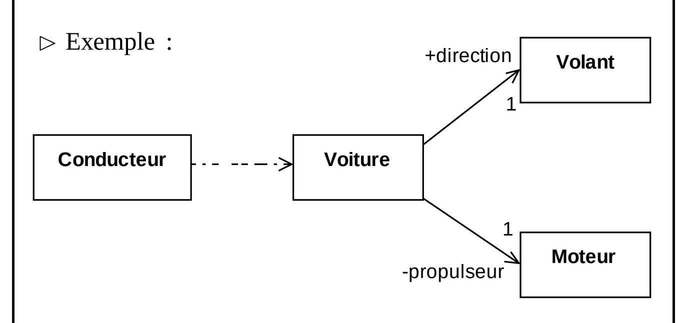


Visibilité d'une association orientée

▷ Une association orientée peut être :

- ◦ privée ;
- ◦ protégée ;
- ◦ public.

▷ Exemple :



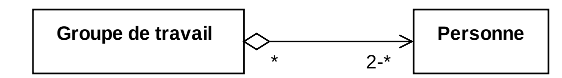
⤳ Le conducteur doit pouvoir manipuler le volant, mais pas le moteur.

Cas particulier d'associations : les agrégations

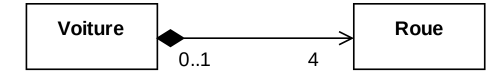
▷ Une agrégation dénote l'idée de « faire partie de ».

▷ On distingue :

- ◦ l'agrégation partagée (ou agrégation)
  Exemple :



- ◦ l'agrégation de composition (ou composition)
  Exemple :



Généralisation / spécialisation

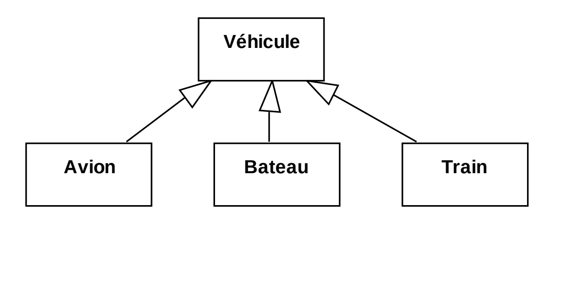
▷ Expression du lien « est-sous-classe-de » entre classes.

▷ Généralisation simple (1 super-classe au maximum) ou multiple.

▷ Toutes les propriétés d'une super-classe sont vraies pour chaque classe fille.

▷ L'invariant d'une sous-classe comprend celui de sa super-classe.

▷ Exemple de généralisation simple :



Contraintes sur la généralisation

UML permet d'ajouter des précisions sur les liens de généralisation. Plusieurs contraintes sont pré-définies :

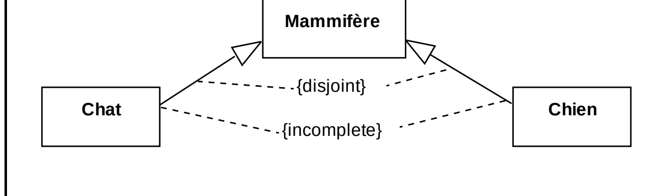
▷ `{overlapping}` ⤳ une classe dérivée peut hériter de plus d'une mère.

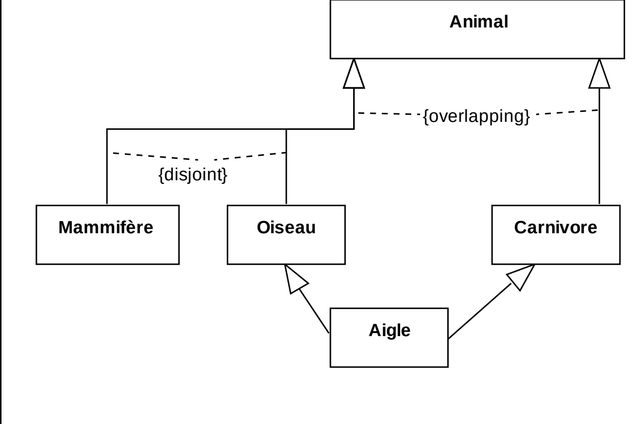
▷ `{disjoint}` ⤳ une classe dérivée NE PEUT hériter que d'une mère.

▷ `{complete}` ⤳ toutes les classes filles sont spécifiées.

▷ `{incomplete}` ⤳ la liste des classes filles est incomplète.

Contraintes sur la généralisation : exemples





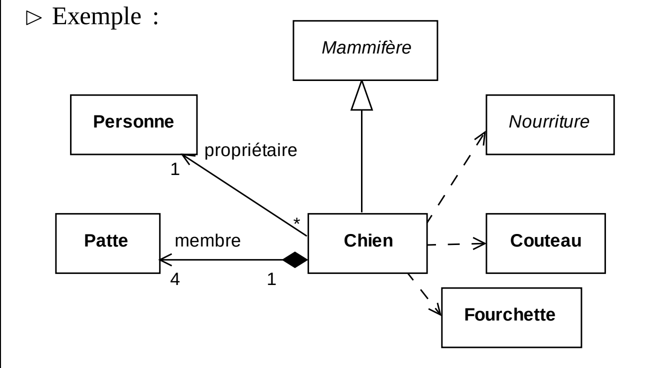
UML : modèle des classes

Liens entre classes : résumé

▷ Trois niveaux pour les liens d'utilisation entre classes :

1. **conceptuel :** liens *forts* et *permanents* entre objets
   ⇝ en UML : association et généralisation ;
2. **contextuel :** paramètre d'une opération
   ⇝ en UML : dépendance ;
3. **opérationnel :** utilisation dans le corps de la méthode
   ⇝ en UML : dépendance.

▷ Exemple :



UML : modèle des classes

Utilisation contextuelle et opérationnelle : exemple

Soit la méthode manger de la classe Chien :

```
manger(Nourriture repas) {
  // Utilise les classes :
  //      couteau, fourchette.
    .
    .
    .
  Couteau.prendre();
  Fourchette.prendre();
  Fourchette.immobiliser(repas);
  Couteau.couper(repas);
    .
    .
    .
}
```

⇝ Utilisation contextuelle de «nourriture» ;

⇝ Utilisation opérationnelle de «couteau» et «fourchette».

UML : modèle des classes

Diagrammes d'instances (ou diagrammes d'objets)

▷ Possibilité d'y mélanger classes et instances.

▷ Désignation des instances :

`nom_de_l'instance : Nom_de_la_Classe`

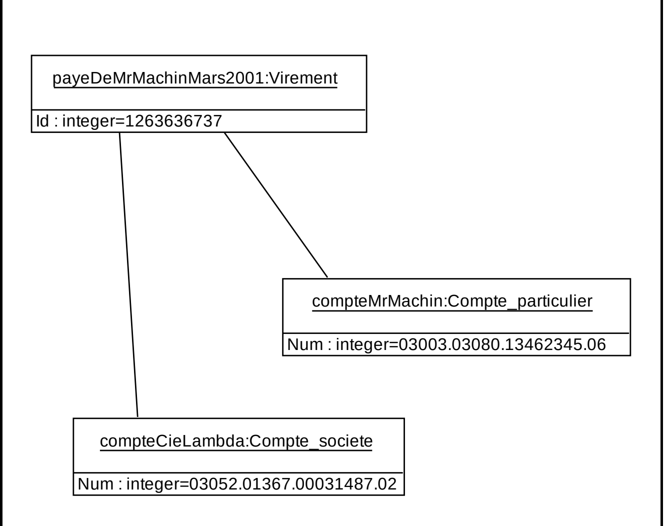
ou bien `nom_de_l'instance`

ou bien `: Nom_de_la_Classe`

▷ Utilisation :

- permet de visualiser les valeurs des attributs ;
- permet de visualiser la totalité des instances lorsqu'on en connaît à l'avance le nombre ;
- permet d'illustrer sur des exemples la pertinence des différents liens ;
- permet de visualiser l'état du système à un instant donné ;
- facilite la compréhension de structures de données complexes (récursives).

Diagrammes d'instances : exemple



Question : il y a plusieurs anomalies dans ce diagramme, lesquelles ?

Règles d'emploi du modèle des classes

▷ Un modèle peut être correct, c'est-à-dire cohérent par rapport aux notations et à la syntaxe, sans qu'il soit pour autant valide (c-à-d. sans correspondre au problème que l'on veut modéliser).

▷ Il ne sera pas surprenant que le modèle objet construit ait besoin de nombreuses révisions.
⇝ Itérer pour clarifier les noms, ajouter des détails, prendre en compte correctement les contraintes structurelles.

▷ Choisir les noms avec soin. Ils sont importants. Ils doivent être précis, descriptifs, non-ambigus.

▷ Documenter systématiquement le modèle des classes. Un diagramme spécifie la structure d'un modèle, mais ne peut pas décrire, notamment, les raisons qui ont poussé à certains choix.

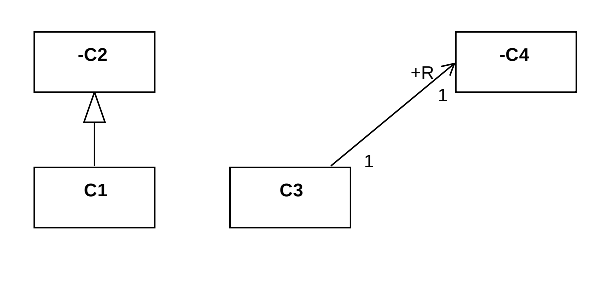
Règles relatives aux packages

▷ Pas de dépendances mutuelles entre 2 packages (ni de cycles dans les liens entre packages)

⇝ Voir les packages comme moyen de structuration strict (en vue d'un découpage du travail de conception / programmation en équipes séparées).

▷ Dans un même package, ne pas ouvrir la visibilité sur les classes privées d'un package (ni par héritage, ni par une association publique).

⇝ (mauvais) Exemples :



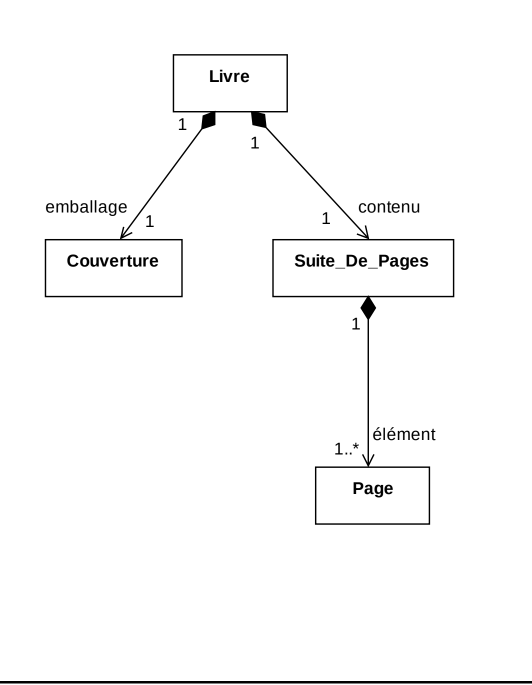
Règles relatives à la définition de classes

▷ Une classe doit correspondre à une certaine « réalité » de la chose à modéliser :
- ne pas faire apparaître de classes *inutiles* ;
- ne pas *dupliquer* de classes ;
- chaque classe doit pouvoir se justifier par les opérations qu'elle fournit de manière exclusive et/ou représenter un concept clairement identifié du système.

▷ Les classes (et les associations) doivent correspondre à des éléments **stables** (par opposition à des états, des dépendances, etc.)

Règles relatives à la définition de classes : exercice

Que penser de cette modélisation ?

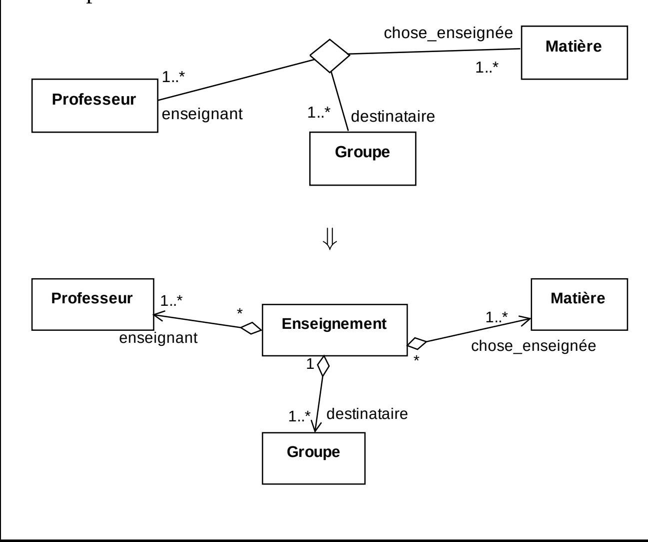


Règles relatives à l'arité des associations

▷ On évitera l'écriture d'associations d'arité > 2.

⇝ Ceci n'est nullement restrictif : toute association n-aire peut se transformer en n associations binaires.

Exemple :



Règles relatives aux attributs

▷ Tout attribut d'une classe devrait être du type d'une classe élémentaire (sinon ⇒ association).

▷ Un attribut doit avoir une signification pour <u>toutes</u> les instances de la classe dans laquelle il est défini.

Règles relatives aux attributs (2)

Identifiant d'une classe : valeur caractérisant un objet.
≡ le sous-ensemble minimal des attributs nécessaires pour distinguer de façon certaine 2 instances quelconque de la classe. (e.g. : num. de sécurité sociale vs nom + prénom).

▷ La valeur d'un attribut n'appartenant pas à l'identifiant I de la classe ne doit pas dépendre (fonctionnellement) :

- ○ d'un attribut n'appartenant pas non plus à I ;
- ○ d'un sous-ensemble strict de I.

Règles relatives à la généralisation

▷ Ne pas factoriser artificiellement des concepts différents.

▷ Rappel : une instance ne peut jamais changer de classe (penser aux états dans ce cas).

▷ Se souvenir que toute classe dérivée hérite de l'ensemble des propriétés de la classe de base.

Règles relatives à la généralisation : exercice

Que penser de ces généralisations ?

Objet_Avec_Nom_Et_Age
+nom : string
+age : integer

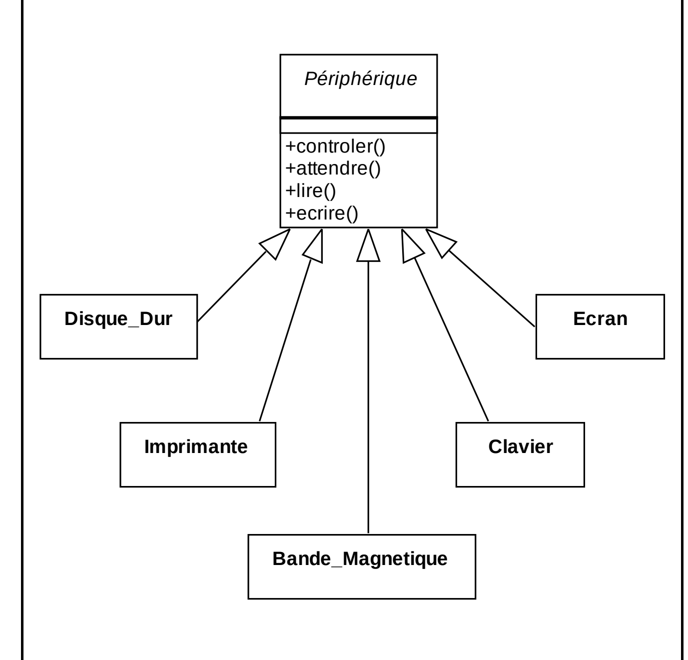
Personne **Planète**

Elément_Gérable
+gérer()

Projet **Compte_Bancaire**

Règles relatives à la généralisation : exercice (2)

Critiquer/améliorer cette modélisation :

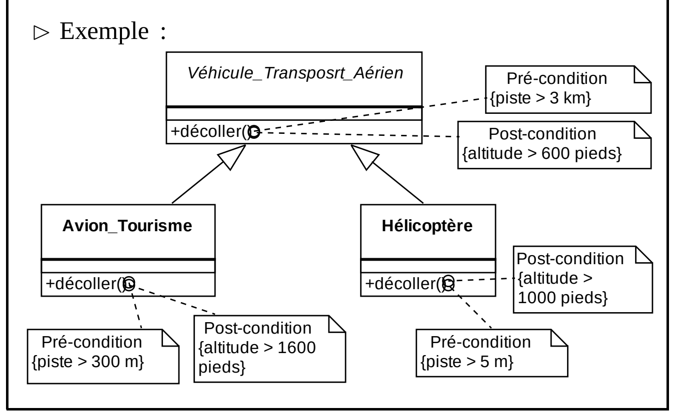


Règles relatives aux pré-/post-conditions

▷ Une classe fille doit permettre l'exploitation de chaque opération dans tous les cas autorisés au niveau de la classe mère...
⟹ maintient ou réduction des pré-conditions.

▷ Aussi, une opération spécialisée doit pouvoir accomplir *au moins* ce qu'accomplit son équivalent dans la classe de base...
⟹ maintient ou augmentation des post-conditions.

▷ Exemple :

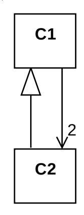


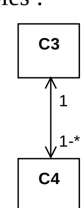
Règles relatives aux orientations des liens entre classes

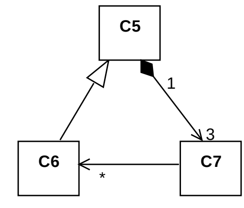
▷ Les orientations mutuelles sont à éviter :
⇝ gène pédagogique ;
⇝ difficultés pour développement, tests et intégration.

▷ Degré de gravité :
- niveau conceptuel : grave ;
- niveau contextuel : gênant ;
- niveau opérationnel : acceptable.

▷ (mauvais) Exemples :



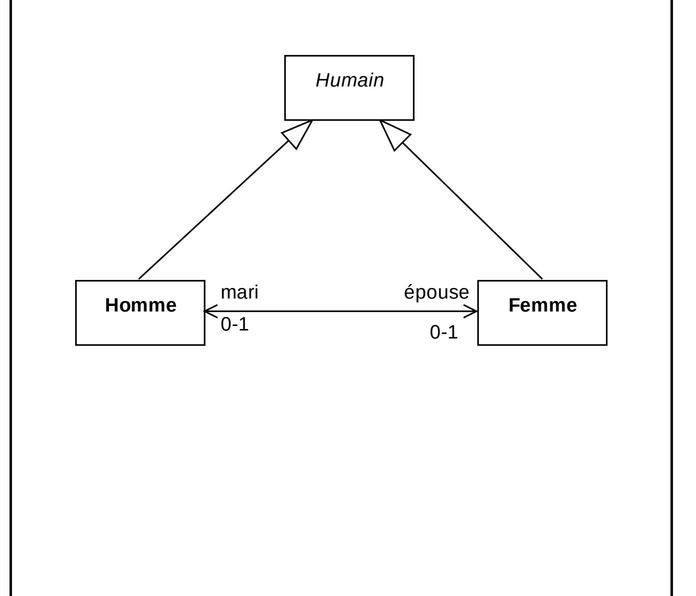




▷ Rappel : dépendances mutuelles entre *packages* strictement interdites.

Orientations mutuelles : exercice

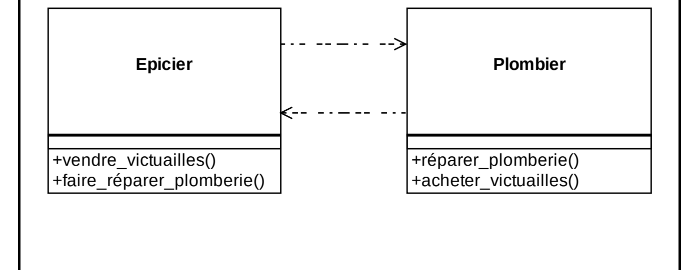
Critiquer/améliorer cette modélisation :



Exercice : le plombier et l'épicier

Comme pour les associations : éviter dans la mesure du possible les utilisations contextuelles mutuelles.

Exercice : critiquer/améliorer cette modélisation



UML : compléments au modèle des classes

1. **Interfaces**.
2. **Classes paramétrables**.
3. **Attributs/associations dérivés et identifiants.**
4. **Responsabilités, etc.**.

Interfaces

▷ Concept similaire aux interfaces Java. Une interface :
⇝ porte un nom ;
⇝ liste des opérations (publiques) ;
⇝ peut lister des signaux (notamment des exceptions) ;
⇝ ne comporte ni attributs, ni associations ;
⇝ ne peut pas être instanciée ;
⇝ sert à spécifier un type abstrait pouvant être implanté par une ou plusieurs classes.

▷ Possibilité de *généralisation* entre interfaces.

▷ Lien d'*implantation* (ou de « *réalisation* ») entre une classe C et une interface I
$\Longrightarrow$ la classe C implante les opérations listées dans I.

▷ Lien de *dépendance* entre une classe C et une interface I
$\Longrightarrow$ classe C utilise le type abstrait défini par I.

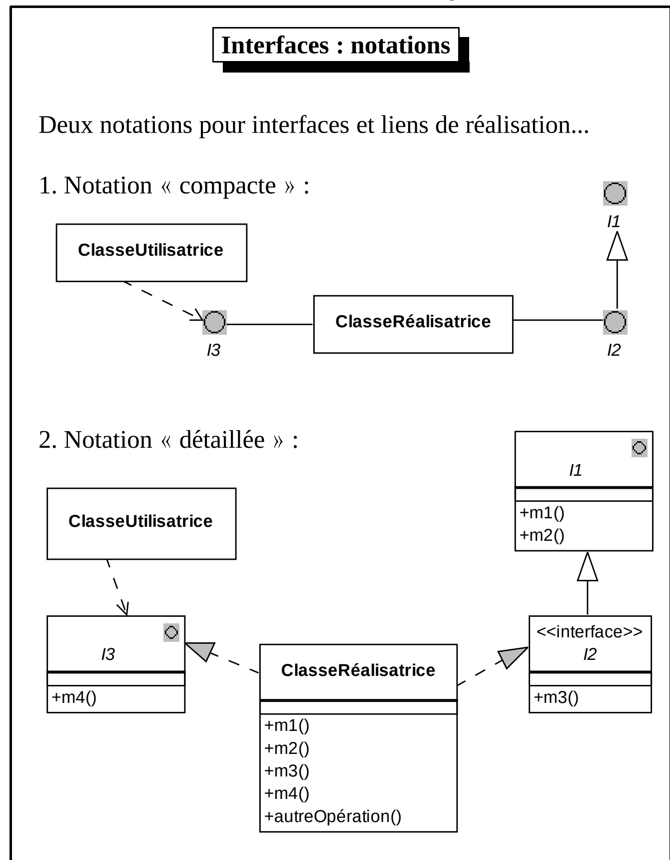
▷ <u>NOTA-BENE :</u> en termes de méta-modèle UML, une interface est une classe avec le stéréotype « interface ».

Interfaces : notations

Deux notations pour interfaces et liens de réalisation...

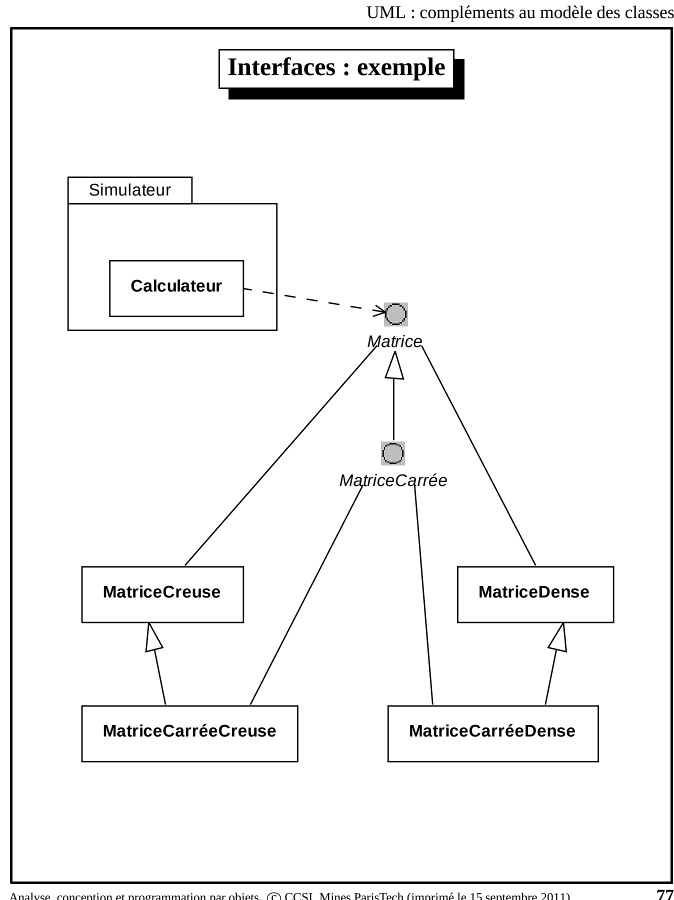
1. Notation « compacte » :

2. Notation « détaillée » :

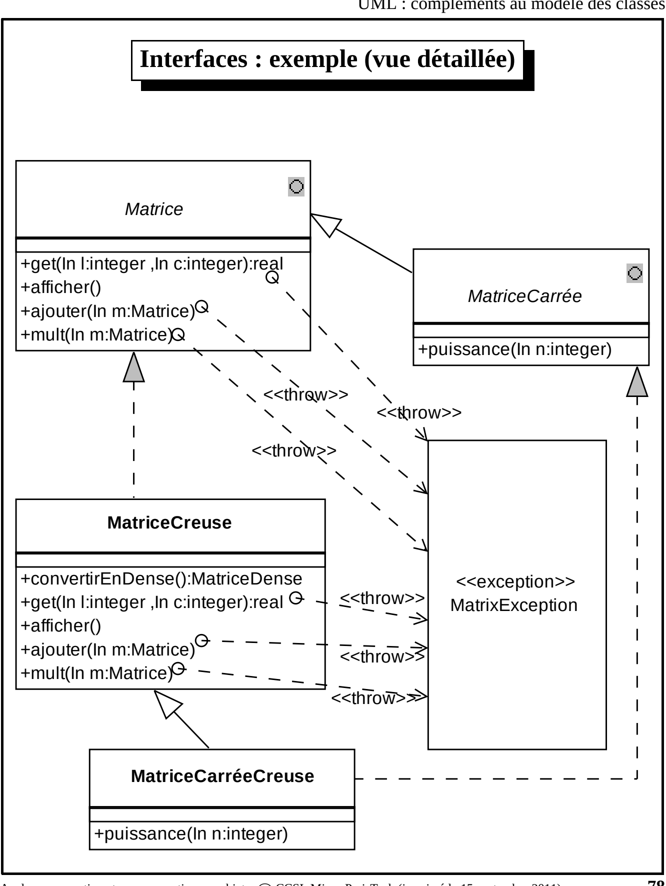




Interfaces : exemple



Interfaces : exemple (vue détaillée)



Résumé : interfaces, classes, classes abstraites

En UML et en Java :

- ▷ une interface équivaut à un *type abstrait* (n'inclut aucune implantation) ;
- ▷ une classe (non-abstraite) équivaut à un *type abstrait accompagné d'une implantation complète* ;
- ▷ une classe abstraite est un concept intermédiaire qui équivaut à *type abstrait accompagné d'une implantation incomplète (ou vide)*.

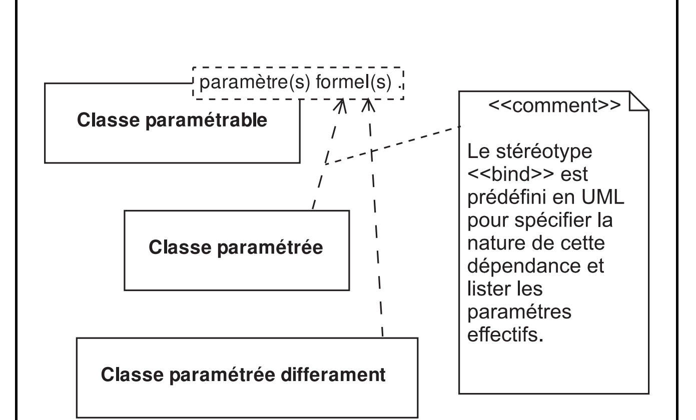
Remarques :

- ▷ Une classe (abstraite ou pas) peut implanter zéro, une ou plusieurs interfaces.
- ▷ Inversement, une même interface peut être implantée par zéro, une ou plusieurs classes.
- ▷ Seules les classes non abstraites peuvent être instanciées.

Résumé : généralisations et liens d'implantation

- ▷ Généralisation entre deux interfaces :
  ⤳ relation de nature sous-type / type entre deux types abstraits.
- ▷ Généralisation entre deux classes :
  ⤳ la sous-classe hérite à la fois le type abstrait et l'implantation correspondant à la classe mère.
- ▷ Lien d'implantation entre une classe (non abstraite) et une interface :
  ⤳ relation de nature type concret / type abstrait ;
  ⤳ la classe (éventuellement via les classes dont elle hérite) inclut (au minimum) le code correspondant à l'ensemble des méthodes définies dans l'interface.
- ▷ Lien d'implantation entre une classe abstraite et une interface :
  ⤳ sémantique moins claire que dans les autres cas (permet de combiner un sous-typage abstrait *et/ou* un début de concrétisation).

Classes paramétrables (ou classes templates)



- ▷ Sémantique identique aux templates C++ et Java.

Responsabilités, ...

En vrac :

- ▷ Responsabilités (d'une classe) : mention (pour les classes d'interface) sous le nom de la classe du, ou des, cas d'utilisation qu'elle prend en charge (cf. transparent 137 sur la recherche des classes d'interface).
- ▷ Contraintes « `Xor` » (ou exclusif) au niveau d'un ensemble d'associations partant d'une classe.
- ▷ Classes/objets actifs : pour modéliser les applications multi-threadées (notation : rectangle à bord gras).
- ▷ Multi-objets : collection d'instances similaires dans un diagramme d'instance...

UML : modèle dynamique

1. **Évènements**.
2. **Diagrammes d’états**.
3. **Diagrammes de séquence**.
4. **Diagrammes de collaboration**.
5. **Diagrammes d’activité**.
6. **Importance des commentaires**.
7. **Résumé et mode d’emploi du modèle dynamique**.

Événements

▷ Occurrence (instantanée) d’une situation donnée significative pour le système.

▷ Les événements peuvent être :

- *internes* au système (émis par un objet) ;
- *externes* (capteur physique, bouton d’une IHM, etc.).

▷ Un événement peut apparaître dans les diagrammes :

- d’états ;
- de séquence, de collaboration, d’activité ;
- de classes (événements « *signaux* » seulement, voir plus loin).

Quatre types d’événements

▷ Événements *asynchrones* :

- **Événement signal** : stimulus *asynchrone*
  ⇝ déclenche réaction chez le destinataire ;
  ⇝ pas de « valeur de retour » possible mais possibilité d’émettre un autre signal en réponse au premier.
- **Événement temporel** : expiration d’une temporisation.
- **Événement modification** : passage à vrai d’une expression booléenne.

▷ Événements *synchrones* :

- **Événement appel** : appel d’une *opération*
  ⇝ déclenche l’opération chez le destinataire ;
  ⇝ « valeur de retour » possible vers l’appelant.

Événements de type « signal »

▷ Un signal est un *objet* émis de manière asynchrone par un objet « émetteur » à destination d’un autre objet « récepteur ».

▷ Un signal peut être d’origine interne ou externe au système.

▷ Exemple de signal interne : une exception (Java).

▷ Exemple de signal externe : une action de l’utilisateur (via l’IHM) ou l’action d’un capteur physique.

▷ En tant qu’objet, un signal doit être instance d’une classe portant le stéréotype « signal » (classe à définir dans le modèle des classes).

Aussi, il peut comporter des attributs et des opérations spécifiques.

▷ Possibilité de visualiser les émissions possibles de signaux dans un diagramme de classe à l’aide d’une dépendance portant le stéréotype « send ».

Exemple de signaux dans un diagramme de classes

<<signal>>
ContextePhysique

<<signal>>
PbHumidité

<<signal>>
PbTempérature
+température : integer

<<signal>>
TropFroid

<<signal>>
TropChaud

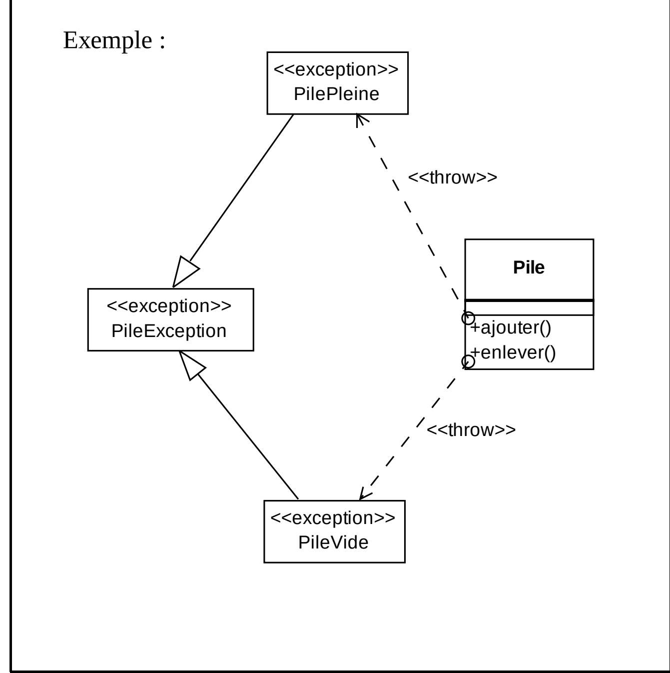
<<signal>>
BeaucoupTropFroid

<<signal>>
BeaucoupTropChaud

Cas particulier de signal : les « exceptions »

▷ Instance d'une classe portant le stéréotype « exception ».

Exemple :



Événements « temporels » (ou time events)

▷ Permet de spécifier le passage d'une certaine quantité de temps.

▷ Instant de référence par défaut : l'entrée dans l'*état* courant (voir plus loin).

▷ Exemples :

◦ `after 30 seconds`

◦ `after 2 ms since exiting EnVeille`
(fait référence à la sortie de l'état de nom `EnVeille`)

Événements « modification » (ou change events)

▷ Permet de représenter un changement d'état ou la satisfaction d'une condition.

▷ La condition qui définit un événement « modification » peut faire référence aux attributs et aux opérations de la classe recevant l'événement (plus ceux des classes visibles depuis cette dernière).

▷ Exemples :

◦ `when temperature > 30`

◦ `when nbInstances = 1000`

◦ `when client.cpt.solde > calculerSeuil()`

Diagrammes d'états (ou statechart diagram)

- Exprime les règles d'utilisation des opérations de la classe ainsi que les aspects réactifs du système :
  - permet de contrôler l'ordre d'exécution des opérations au sein d'une classe ;
  - permet de spécifier que certaines opérations ne peuvent s'appliquer que lorsque l'objet est dans un certain état et/ou certaines contraintes sont satisfaites ;
  - permet de tenir comptes des occurrences d'événements.
- Il est défini par un *automate d'états fini*, c-à-d. un graphe orienté dont les noeuds sont des *états* et les arêtes des *transitions*.

Diagrammes d'états : principes

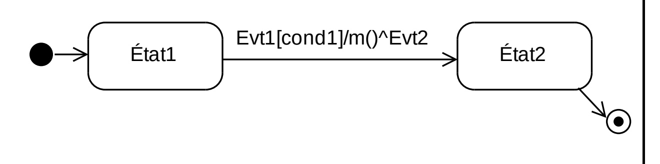
- Un diagramme d'états est généralement associé à une classe (peut aussi être relatif à un use case ou au système global).
- Une classe peut avoir plusieurs diagrammes d'états.
- Un diagramme d'états doit être *déterministe* : pour chaque état les transitions sortantes doivent correspondre :
  - à des événements différents ;
  - ou alors être gardées par des conditions incompatibles.

Diagrammes d'états : états et transitions

Les diagrammes d'états sont définis par :

- des **états** : situations caractéristiques et stables du fonctionnement des instances de la classe.
  - Cas particuliers : les états **initial** et **final**.
- des **transitions** : manières dont un objet passe d'un état à un autre... Une transition peut :
  - correspondre à l'arrivée d'un événement ;
  - avoir une condition de garde ;
  - donner lieu à l'exécution d'une opération de la classe et/ou l'émission d'un événement.

Notation :



Diagrammes d'états : exemple

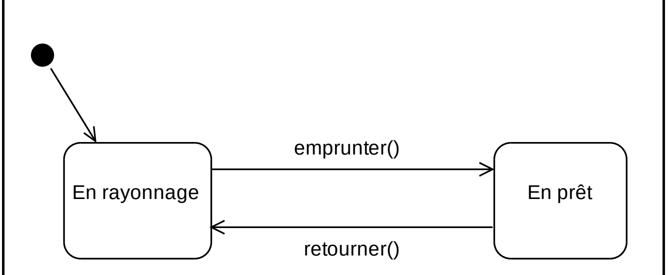


Diagramme d'états de la classe `Livre`

Remarquer :
- l'état initial ;
- les 2 autres états ;
- les 2 transitions, chacune étant déclenchée par un événement de type appel.

Diagrammes d'états : deuxième exemple

EnVeille
tropFroid(températureSouhaitée)
tropChaud(températureSouhaitée)
températureOK
températureOK
Chauffage
Refroidissement
tropChaud(températureSouhaitée)
tropFroid(températureSouhaitée)

Diagramme d'états de la classe `Thermostat`

Diagrammes d'états : troisième exemple

Vide
débutEmbarquement/embarquerPassagers()
/débarquerPassagers()
Chargé
autorisationRejoindreLaPiste/rejoindrePiste()^demandeAutorisationDécollage
PrêtADécoller
/rejoindreAérogare()
autorisationDeDécollage/décoller()
FinAttérrissage
EnVol
autorisationDAttérissage/atterir()
changerCap()

Diagramme d'états de la classe `Avion`

Diagrammes d'états : précisions sur les états

▷ Pour chaque état, possibilité de spécifier :

- une action exécutée à l'entrée dans l'état ;
- une action exécutée à la sortie de l'état ;
- une *activité* à effectuer tant qu'on reste dans l'état ;
- des transitions internes (qui ne déclenchent donc pas les actions ci-dessus) ;
- des événements dont le traitement est « retardé ».

▷ Exemple :

Appel en attente
entry/incrémenterCompteurClientsEnAttente()
exit/decrémenterCompteurClientsEnAttente()
do/jouerMusiqueDouce()
after(90 seconds)/jouerMessageExcusePourAttente()
toucheClavier(n)/defer

Diagrammes d'états : états composites

▷ Permet de simplifier les automates d'états en considérant des *états* composés de *sous-états*.

▷ Le diagramme d'état correspondant à un état « composite » doit comporter au maximum un état initial et un état final.

Diagrammes d'états : quatrième exemple

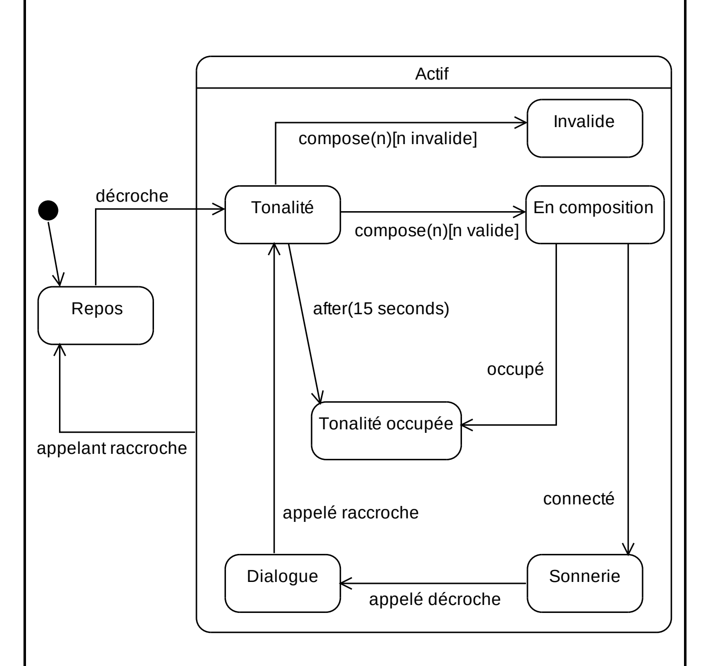


Diagramme d'états de la classe `Téléphone` avec visualisation du contenu de l'état composite `Actif`

Scénarios (ou diagrammes d'interactions)

▷ Exemples de traitement au sein du système.

▷ Associés à un package, une classe, ou un cas d'utilisation.

▷ S'expriment sous la forme d'une séquence d'envois de messages et/ou signaux entre des objets (objets concrets ou prototypes).

▷ Diagrammes utiles pour :
- vérifier la complétude de l'analyse, mettre en évidence des incohérences, etc. ;
- documenter l'analyse, illustrer le fonctionnement d'une partie du système ;
- préparer des jeux de test.

Diagramme de séquence vs de collaboration

Deux types de diagrammes *sémantiquement équivalents* disponibles en UML pour exprimer des scénarios :

▷ les **diagrammes de séquence** pour mettre en évidence la dimension *temporelle* dans l'enchaînement des interactions entre objets ;

▷ les **diagrammes de collaboration** pour mettre en évidence comment les objets *coopèrent* pour mener à bien le scénario.

Diagramme de séquence : notations

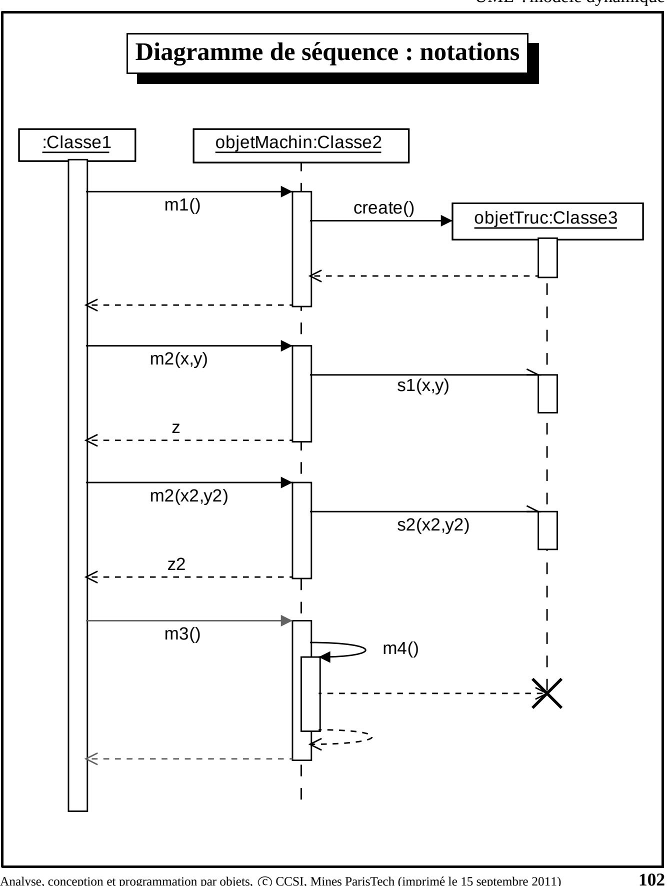


Diagramme de séquence : exemple et exercice

Critiquer/améliorer le scénario suivant :

Jean:Employé
:Patron
salaireDeJean:
demanderAvance(4000)
donnerAvance(2000)
2000
demanderAugmentation()
0
donnerDémission()
rembourser(2000)
rembourserAvance(2000)

Diagramme de séquence : autre exemple

:Usager
:Téléphone
:Standard
décrocher()
sonnerTonalité()
composer(9294)
demander(9294)
sonnerAttenteDeConnexion()
occupé()
sonnerOccupé()
raccrocher()

Diagramme de séquence : exercice

Critiquer/améliorer le diagramme suivant :

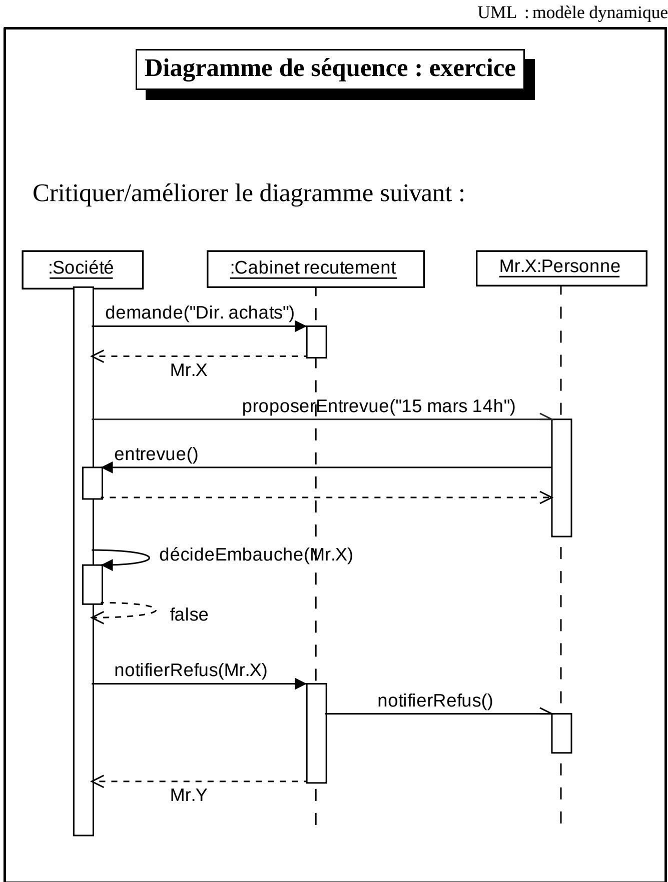
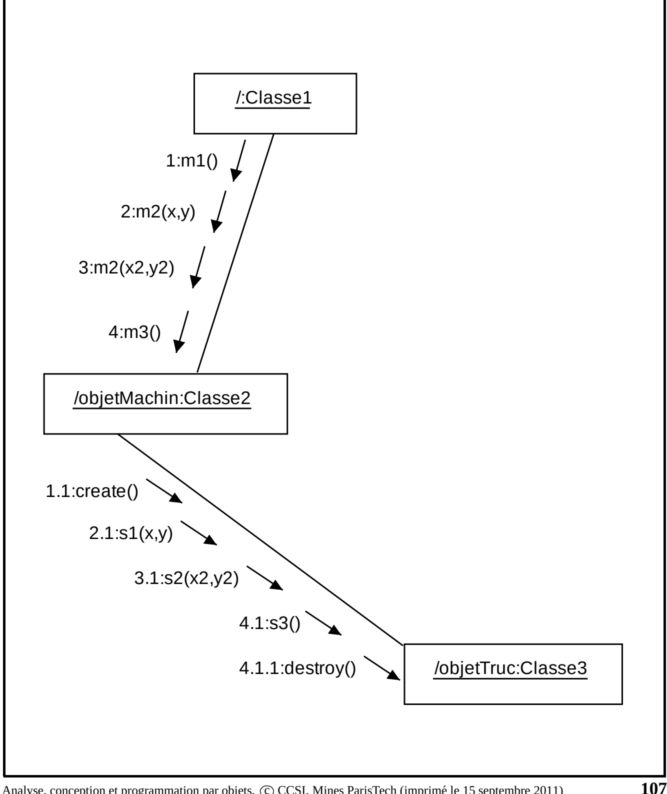


Diagramme de séquence : notations pour switch & while

i1:
i2:
i3:
[CONDITION] m1()
[non CONDITION] m2()
[CONDITION1] m3()
[CONDITION2] m4()
[CONDITION3] m5()
* [CONDITION] m6()

Diagramme de collaboration : notations



Diagrammes d'activités

▷ Graphe orienté associé à une activité.

▷ Permet de spécifier le fonctionnement interne d'une *activité* en la décomposant en :

- ○ actions (e.g. envoi de signal, appel d'opération, création/destruction d'objet, évaluation d'une expression) ;
- ○ sous-*activités* (chacune pouvant être spécifié à son tour par un diagramme d'activités).

Rappel : les activités sont décomposables et interruptibles alors que les actions ne le sont pas.

Diagrammes d'activités : utilisation

Deux contextes d'utilisations distincts :

1. Macroscopique

   ▷ Modélisation d'un flux de traitements (workflow) :

   - ○ permet de décomposer les activités du point de vue des *acteurs* du système
     ⇝ détailler/expliciter les activités métiers ;
   - ○ diagramme associé typiquement à un cas d'utilisation (éventuellement à un package, une classe ou une collaboration.)

2. Microscopique

   ▷ Modélisation d'une opération (flowchart) :

   - ○ visualise la séquence d'instructions correspondant au corps de la méthode
     ⇝ « programmation visuelle » ;
   - ○ diagramme associé typiquement à une opération.

Diagrammes d'activités : principes et notations

▷ Actions et sous-activités se représentent de la même manière en tant que sommet du graphe tandis que les arcs indiquent l'enchaînement depuis l'état initial jusqu'à l'état final.

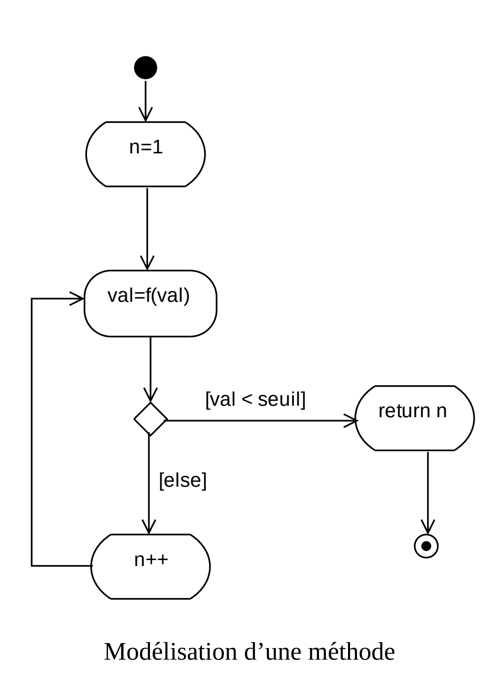
▷ Les sous-activités peuvent comporter des champs `entry` et `exit`.

▷ Contrairement aux diagrammes d'états, les transitions s'effectuent dès que l'action ou sous-activité précédente est terminée.

▷ Possibilité d'inclure les instances créés, modifiées ou détruites au sein de l'activité en dessinant un lien de dépendance vers l'instance depuis l'élément du diagramme d'activité qui l'a créé, modifiée ou détruite.

▷ Notations spécifiques pour :
- branchement conditionnel ;
- concurrence (fork & join) ;
- partionnement en lignes d'eau (ou « swimlane »).

Diagramme d'activités : exemple



Modélisation d'une méthode

Diagramme d'activités : autre exemple

Consommateur
Serveur
Cuisinier
Choisir table
Apporter carte
[arrivée nouveau client]
Attente
[client prêt à commander]
Refléchir
Demander choix
Passer commande
Noter commande
Préparer commande

Modélisation d'un workflow

Importance des commentaires

Outre les diagrammes vus jusqu'ici (et afin de préciser le mieux possible la dynamique du système) :

▷ Penser à décrire les traitements en attachant des descriptifs textuels aux opérations complexes. Ces descriptifs peuvent être de différentes sortes :

1. **Summary :** bref résumé (e.g. 1 ligne) ;
2. **Description :** spécification/explication détaillée du traitement ;
3. **Comment :** commentaire technique (destiné au concepteur/programmeur) .

▷ Prendre soin d'utiliser exactement les termes employés pour la description du modèle (nom des classes, rôle des associations, etc).

Résumé et mode d'emploi du modèle dynamique

1. Si besoin, à titre documentaire, expliciter le contexte général (*métier*) à travers un (ou plusieurs) diagramme(s) d'activités.
2. Expliciter les *interactions utilisateur–système* à travers un (ou plusieurs) scénario(s) macroscopique(s).
3. Identifier les classes ayant des comportements systématiques, réagissant à des conditions particulières. Pour ces classes définir un (ou plusieurs) diag. d'états.
4. Préparer un jeu de scénarios *détaillés* pour tester / illustrer la faisabilité des principales fonctionnalités en termes d'enchaînements de méthodes (diag. de séquence et/ou diag. de collaboration).
5. Si besoin, et à titre documentaire, spécifier le *fonctionnement* des principales ops. de chaque classe :
   – à travers un diag. d'activité ;
   – et/ou un descriptif textuel.

UML : modèle fonctionnel

1. **Acteurs**.
2. **Cas d'utilisation**.
3. **Diagrammes de cas d'utilisation**.
4. **Relations entre cas d'utilisation**.

Acteurs

▷ Un acteur est une entité *extérieure* au système et amenée à interagir avec lui.

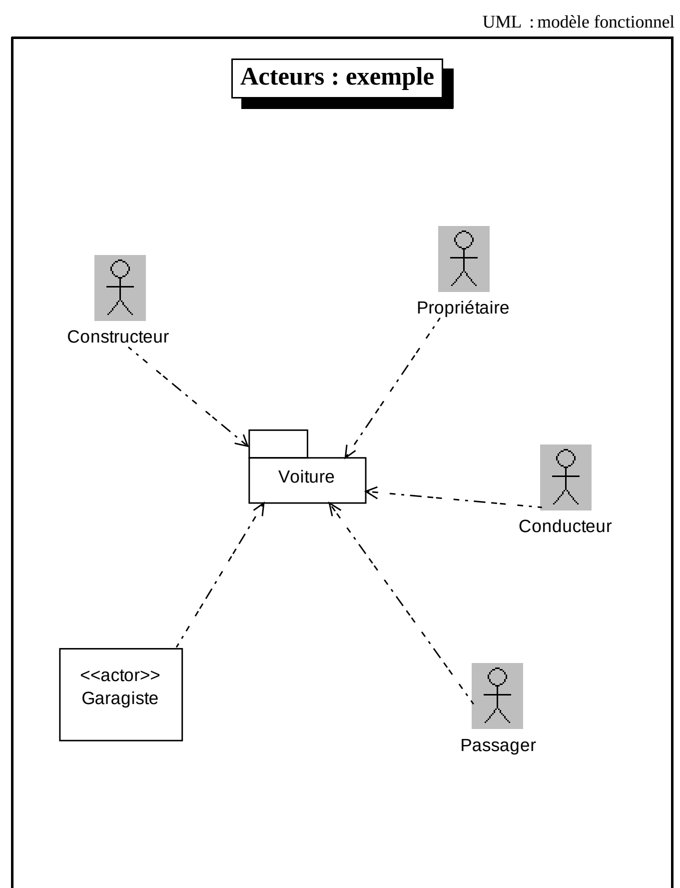
▷ Un acteur est une classe (avec stéréotype « actor »).

▷ Un acteur peut représenter un utilisateur direct du système (humain) ou un autre système.

▷ On distingue :

- les acteurs *primaires* (ceux pour qui le système est directement prévu) ;
- les acteurs *secondaires* (ceux dont l'existence est la conséquence des besoins des utilisateurs primaires).

Acteurs : exemple



Cas d'utilisation (ou use cases)

▷ Un cas d'utilisation est une *manière particulière* d'utiliser le système.

▷ Un cas d'utilisation correspond à une séquence d'interactions entre le système et un ou plusieurs acteurs.

▷ Un jeu de cas d'utilisation permet de décrire de manière informelle le service rendu par le système dans sa globalité, autrement dit, ils fournissent une *expression fonctionnelle* du système.

▷ Un cas d'utilisation porte un nom.

▷ Notation : un ellipse autour du nom du cas d'utilisation.

▷ Les *diagrammes de cas d'utilisation* permettent de visualiser les liens entre acteurs et cas d'utilisations, ainsi que les liens entre cas d'utilisation.

Diagramme de cas d'utilisation : exemple

Voiture

Se déplacer

Ecouter de la musique

Téléphoner

Changer les bougies

Faire la vidange

Conducteur

Garagiste

Relations entre cas d'utilisation

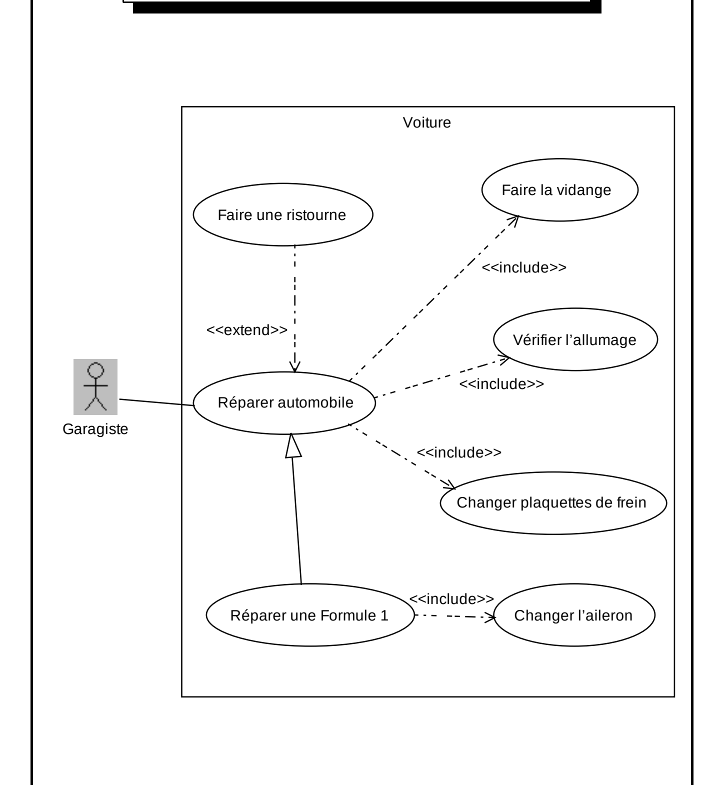
▷ **Inclusion :**
⇝ permet une approche « top-down » de la spécification des besoins utilisateurs (indique qu'un cas d'utilisation se décompose en plusieurs autres).

▷ **Généralisation :**
⇝ permet de définir un cas d'utilisation comme un cas particulier d'un autre plus général et/ou plus abstrait.

▷ **Extension :**
⇝ permet la gestion des « variantes »
– pouvant être ajoutées (ou pas) à une fonctionnalité initiale mais qui ne sont pas strictement nécessaires à la complétion du cas d'utilisation de départ,
– dont la présence ou l'absence dépend, par exemple, du matériel utilisé pour le déploiement du système.

<u>NOTA-BENE :</u> penser au critère de « sous-partie strictement nécessaire » pour discriminer entre relations d'inclusion et d'extension.

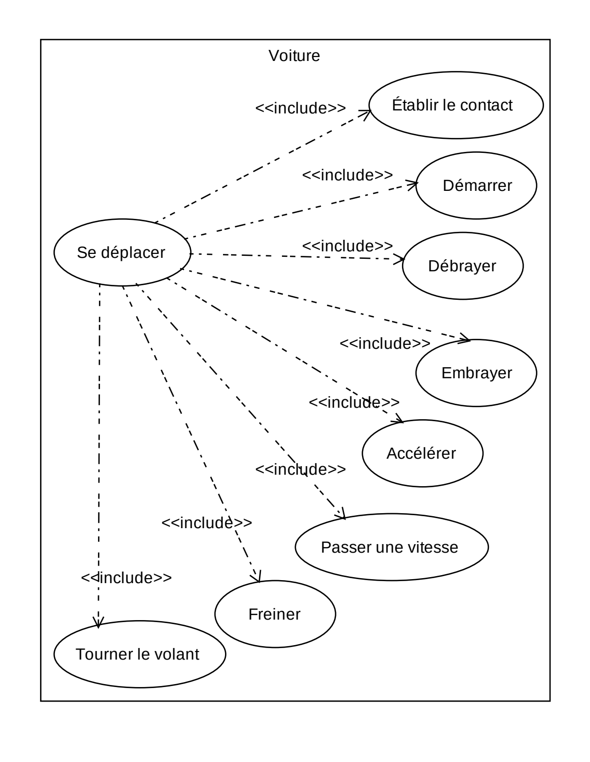
Relations entre cas d'utilisation : exemple



Double intérêt de la relations « include »

▷ Parmi les trois sortes de relations entre cas d'utilisation, la relation « include » est la plus utile car elle permet tout à la fois de *hiérarchiser* et de *factoriser* les cas d'utilisation.

Exemple de hiérarchisation par la relation « include »



Exemple de factorisation par la relation « include »

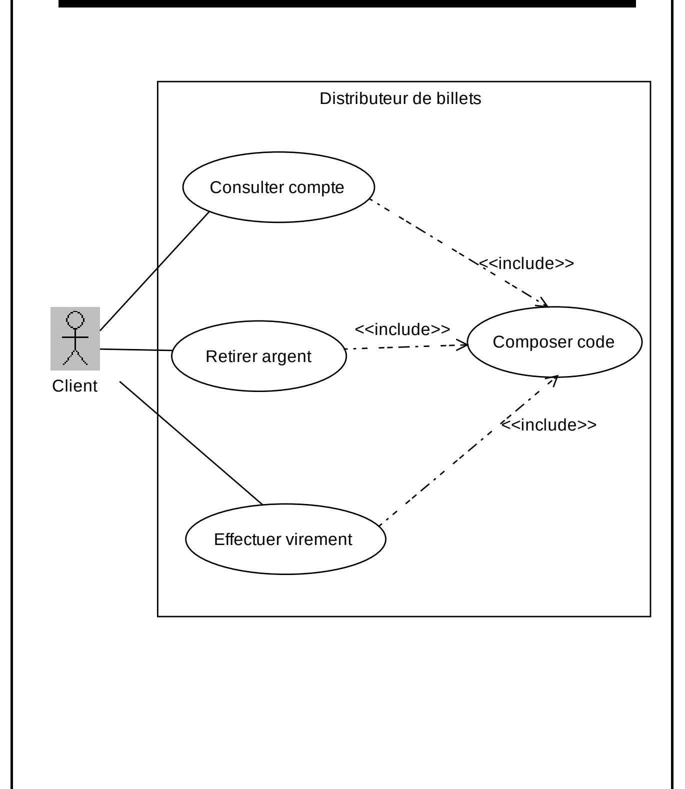
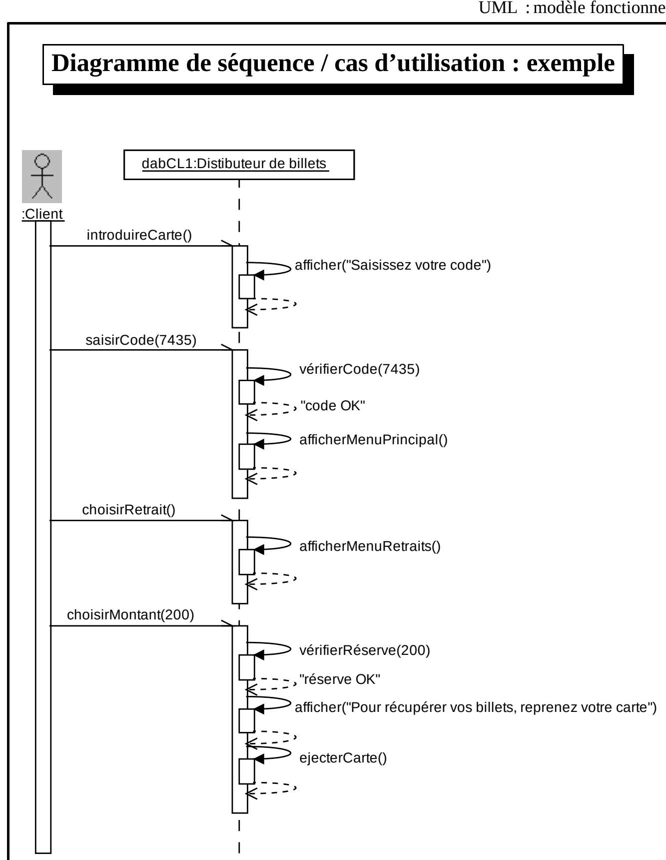


Diagramme de séquence / cas d’utilisation : exemple

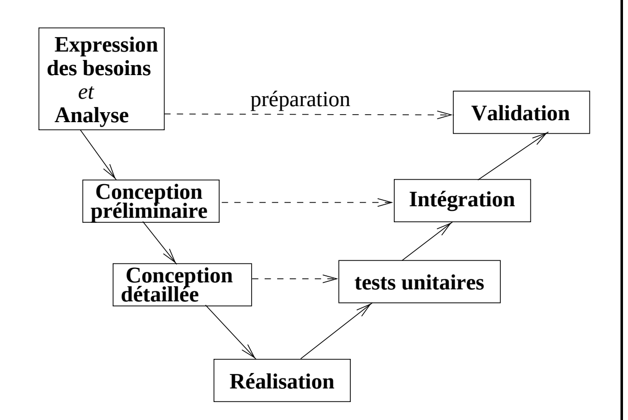


Méthodologie pour l’analyse et la validation

1. **Rappel : le cycle en V**.
2. **Méthodologie : introduction.**
3. **Les étapes de l’analyse**.
4. **Préparation de la validation**.
5. **Résumé : pertinence des différents éléments d’UML**.
6. **Document d’analyse**.

Rappel : le cycle en V

▷ Mise en parallèle des phases de construction et de vérification :



▷ Cycle bien adapté aux « petits » projets (au-delà : penser au cycle en spirale).

Méthodologie : introduction

▷ Méthodologie (ou software development process) :

- démarche menant de la spécification d’un besoin à la réalisation d’un logiciel (ou à la modification d’un logiciel existant) ;
- comporte de nombreux aspects : techniques mais aussi gestionnaires (planification du développement, optimisation des coûts et des délais, maîtrise des risques).

▷ Guide les différents intervenants
⇝ *équilibre* à trouver entre :

- des *contraintes* destinées à assurer la maîtrise des coûts, des délais et le respect des spécifications ;
- la *créativité* indispensable, notamment, lors des phases d’analyse et de conception.

Méthodologie : contenu

Répond aux questions suivantes :

▷ Quelles sont les différentes phases du cycle de vie et comment s’enchaînent-elles ?

▷ Comment s’y prendre pour identifier les besoins, structurer le système en classes et packages, planifier les moyens à mettre en oeuvre pour chaque phase, ... ?

▷ Quand utiliser quel diagramme (plus quels stéréotypes et/ou annotations) et pour quoi faire ?

▷ Quelles sont les productions dues à l’issue de chaque phase : quels documents, logiciels et/ou données ?

▷ Qui réalise quoi dans chaque phase ?

▷ ...

Méthodologie et UML

▷ UML n’inclut pas de méthodologie.

▷ Toutefois,

- UML suggère l’utilisation d’une méthodologie basée sur les *cas d’utilisation* ;
- les auteurs d’UML proposent la méthodologie « RUP » (Rational Unified Process).

▷ Il faut en fait distinguer :

- le langage de modélisation UML ;
- les règles d’utilisation des différents modèles ;
- les méthodologies génériques comme l’UP et ses déclinaisons : RUP, EUP, ... ;
- une méthodologie « ad hoc » (adaptée aux besoins spécifiques d’une équipe pour un projet donné.)

▷ Principales méthodologies compatibles UML :

- « générales / génériques » : UP/RUP/..., Catalysis, ... ;
- « agiles » : XP, Scrum, Crystal, ... (cf. bibliographie).

L'analyse (en 8 étapes)

Huit étapes (pas forcément séquentielles) :

A1. Expression des besoins.

A2. Établir le dictionnaire.

A3. Identifier et définir les classes d'interface.

A4. Structurer le modèle.

A5. Construire les premiers scénarios.

A6. Vérifier la cohérence de l'analyse préliminaire.

A7. Détailler le modèle.

A8. Vérifier la qualité de l'analyse détaillée.

A1 : Expression des besoins (Requirements Elicitation)

▷ Identifier les utilisateurs du système (au sens d'*acteurs* UML : entité externe au système et amenée à interagir avec lui).

▷ Pour chaque type d'utilisateur :

1. Identifier les besoins fondamentaux : ceux dont la satisfaction motivent la réalisation du système.
2. Identifier les besoins opérationnels : ensemble des actions que l'utilisateur devra effectuer pour satisfaire chacun de ses besoins fondamentaux.

▷ Modéliser les utilisateurs en termes d'acteurs.

▷ Modéliser les besoins par des cas d'utilisation.

▷ Décrire (brièvement) sous forme textuelle chaque cas d'utilisation.

▷ Identifier aussi les besoins *non-fonctionnels*
⇝ vitesse des traitements, capacité d'accès simultané, etc.

Exercice : l'ascenceur

Soit un ascenceur dans un petit immeuble d'habitation de 5 étages plus 2 niveaux en sous-sol. Quels sont :
– les *utilisateurs* ?
– les *besoins fondamentaux* ?
– les *besoins opérationnels* ?

Précisions sur l'analyse fonctionnelle

▷ Utiliser le vocabulaire du client exclusivement :

- pas de termes typiquement informatiques (classe, instance, attribut, association, généralisation, etc.) ;
- l'ensemble de l'analyse fonctionnelle doit être facilement compréhensible par le client.

▷ Fournir une vue strictement *externe* du système à réaliser (on ne s'intéresse donc pas encore à son architecture).

▷ Clarifier, à la manière d'un contrat, ce que le système doit faire et ce qu'il ne doit pas faire.

A2 : Établir le dictionnaire (ou « glossaire »)

Nom	**Définition**	*Traduction UML*	*Nom informatique*

▷ Préciser les noms et les définitions propres à l'application (remplir les 2 premières colonnes).

▷ Justifier toute absence de terme.

▷ Retirer les synonymes et qualifier les homonymes.

▷ Valider le dictionnaire avec le « client ».

Précision sur le dictionnaire

▷ Moyen pour **établir** et **figer** la terminologie du domaine de l'application.

▷ Point d'entrée et référentiel initial de l'application.

▷ Outil de communication avec les clients.

▷ Outil pour assurer la cohérence entre analyse, conception et réalisation.

▷ Rangement par ordre alphabétique.

▷ Éventuellement, constitué de deux parties :

1. dictionnaire des notions,
2. dictionnaire des actions.

A3 : Identifier et définir les classes d'interface

▷ Point de départ : cas d'utilisation et dictionnaire.

▷ Identifier les classes (et les opérations associées) qui satisfont les besoins opérationnels des utilisateurs...
⇝ ces classes sont les classes d'interface (ou boundary classes).

▷ Poursuivre dans cette voie en considérant individuellement chaque classe C identifiée :

- C peut-elle réaliser ses services de manière autonome ? Sinon de quoi a-t-elle besoin ? Quelle autre classe pourrait satisfaire ce besoin ?
- quelles opérations supplémentaires la classe C peut-elle fournir pour constituer une entité cohérente ?

A4 : Structurer le modèle

▷ Définir les associations entre classes.

▷ Définir les généralisations entre classes.

▷ Indiquer les classes « abstraites ».

▷ Définir l'organisation en packages.

A5 : Construire les premiers scénarios

▷ Point de départ :

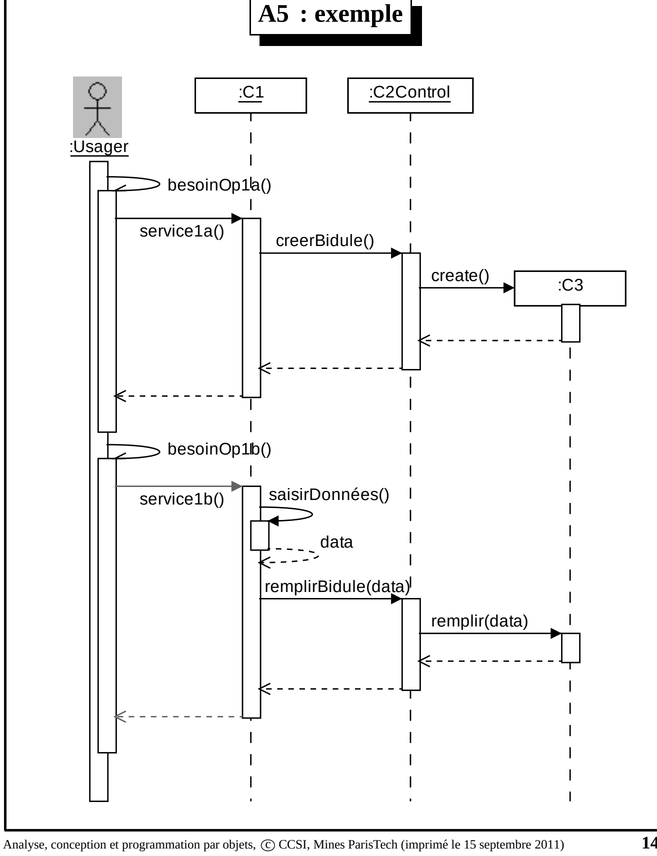
1. les cas d'utilisation ;
2. les classes d'interface identifiées en A3.

▷ Définir les scénarios sous forme de diagrammes de séquences (et/ou de collaboration).

▷ Remarque : lors de l'analyse, les scénarios sont pertinents à la fois pour :

- préciser le cahier des charges et les use cases ⇝ scénarios utilisables *aussi* pour la validation ;
- détailler les appels de méthodes entre les classes ⇝ scénarios utilisables *aussi* pour les phases suivantes (conception, réalisation et tests unitaires).

A5 : exemple



A6 : Vérifier la cohérence de l'analyse préliminaire

Vérifier la cohérence entre :

1. le contenu du dictionnaire ;
2. les cas d'utilisation ;
3. les scénarios ;
4. les éléments modélisés jusqu'ici dans les différents diagrammes de classe.

Il s'agit notamment de vérifier la cohérence entre la vision externe (les cas d'utilisation) et interne (diagrammes de classe) du système.

Attention : il est fréquent de devoir « boucler » plusieurs fois sur les étapes A1 à A6 avant de parvenir à une analyse préliminaire satisfaisante.

A7 : Spécifier en détail le modèle

▷ Compléter la spécification de chaque classe :

- attributs et opérations ;
- pour chaque opération : liste des paramètres et type de retour ;
- pour chaque attribut et paramètre : son type ;
- pour chaque association : orientation, cardinalité(s) et rôle(s) ;
- dépendances vers d'autres classes ;
- caractère « de classe » des attributs, ops., assocs. ;
- ...

▷ Spécifier, lorsque c'est nécessaire :

- les invariants des classes et des packages ;
- les automates d'états des classes ;
- les pré-/post-conditions des opérations ;
- les diagrammes d'activités des opérations.

A7 : Spécifier en détail le modèle (suite)

▷ Appliquer les règles d'emploi du modèle des classes (suppression des cycles, etc.).

▷ Ajouter les précisions documentaires utiles à la clarté de la modélisation.

... Par contre, il est *inutile* en phase d'analyse de spécifier :

▷ les visibilités ;

▷ les accesseurs (pour les attributs et les associations) ;

▷ les constructeurs (sauf si ils sont non-triviaux).

A8 : Vérifier la qualité de l'analyse détaillée

▷ Vérifier :

○ la correction,

○ la completude,

○ et la consistance de la modélisation

(sinon retour en A1 ou A7).

⇝ penser à utiliser l'outil si il permet la recherche automatique de défauts dans la modélisation.

▷ Compléter le dictionnaire en précisant les formes de modélisation UML des termes et les noms informatiques utilisés (colonnes n° 3 et n° 4) :

Nom	*Définition*	**Traduction UML**	**Nom informatique**

Qualité de l'analyse détaillée : reviewing

Liste (non-exhaustive) de questions à se poser :

▷ tous les termes du dictionnaire sont-ils bien définis et compréhensibles par l'ensemble des intervenants ?

▷ Pour chaque classe : est-elle nécessitée directement ou indirectement par un cas d'utilisation ? Si oui, dans quel cas d'utilisation des instances de cette classe sont-elles crées, modifiées, détruites ?

▷ Pour chaque attribut : quel est son type ? quand est-il initialisé ? modifié ?

▷ Pour chaque association : quand est-elle traversée ? sa multiplicité et son orientation sont-ils justifiés ?

▷ Y a t-il des entités portant le même nom, et si oui pourquoi (dénotent-elles bien des notions identiques) ?

▷ Toutes les entités sont-elles bien décrites avec un même niveau de détail ?

▷ les cas d'erreurs sont-ils tous décrits et traités ?
(idem pour phases d'initialisation et d'arrêt du système)

Précisions sur l'analyse détaillée

▷ Les objectifs sont assez opposés à ceux de l'analyse fonctionnelle :

○ utilisation d'un langage davantage informatique (cas d'utilisation et scénarios mais aussi diagrammes de classe, diagrammes d'états, etc.) ;

○ obtention d'une vue à la fois *externe* et *interne* du système à réaliser ;

○ niveau de détail dans la compréhension des besoins qui va bien au delà de ce qui intéresse généralement l'utilisateur (c-à-d. l'aspect contractuel de l'analyse fonctionnelle).

▷ Ne doit pas comporter de redondances, d'incohérences ou d'imprécisions
⇝ en effet, il s'agit de bien mesurer la complexité *interne* de la chose à réaliser et de délimiter l'espace des décisions à prendre lors de la phase suivante (conception).

Préparation de la validation

▷ Définir une stratégie en vue de la validation :
- ○ par type d'acteurs ;
- ○ par cas d'utilisation ;
- ○ ou par packages.

▷ Compléter et finaliser les scénarios de validation (et, le cas échéant, les jeux de tests correspondant).

▷ Planifier la phase de validation en termes de temps et de moyens nécessaires.

Résumé : pertinence des différents éléments d'UML

Phase	*An.*	*Co.*	*Ré.*	*In.*	*Va.*
Modèle fonctionnel :					
Acteur et use case	●●●	●	●	●	●●
Modèle des classes :					
Package	●●	●●●	●●●	●●●	●●
Classe	●●●	●●●	●●●	●●●	●●●
Attribut	●●●	●●●	●●●	●	●
Opération	●●●	●●●	●●●	●●●	●●●
Association	●●●	●●●	●●●	●●	●
Visibilité		●●●	●●●	●●	
Invariant	●●	●	●	●●	●●
Pré-/Post-condition	●●	●●	●	●●	●●
Modèle dynamique :					
Diagramme d'états	●●	●●	●	●●	●●
Scénario	●●●	●●		●●	●●●
Diagramme d'activités	●	●	●		●

● parfois utile ●● très utile ●●● indispensable

Document d'analyse : plan type

1. Présentation générale
   - ▷ Situation et objectifs, documents de référence, etc.
2. Spécification préliminaire
   - ▷ Dictionnaire.
   - ▷ Représentation globale de l'application.
   - ▷ Description résumée.
3. Définition des utilisateurs
   - ▷ Cas d'utilisation (et acteurs), exemples (scénarios).
4. Spécification détaillée
   - ▷ Définition du modèle des classes (par package) :
     - package p_1 :
       classes C_1, C_2, ... C_n
       chacune accompagnée le cas échéant de ses :
       invariants, diagramme d'état, ...
     - package p_2 :
       ...

UML : modèle d'implantation

1. **Composant**.
2. **Diagramme de composants**.
3. **Noeud et liaison**.
4. **Diagramme de déploiement**.
5. **Directives**.
6. **Hypergénéricité**.

Composant

▷ Élément physique du système
(codes source ou exécutable, fichier de données, ...)

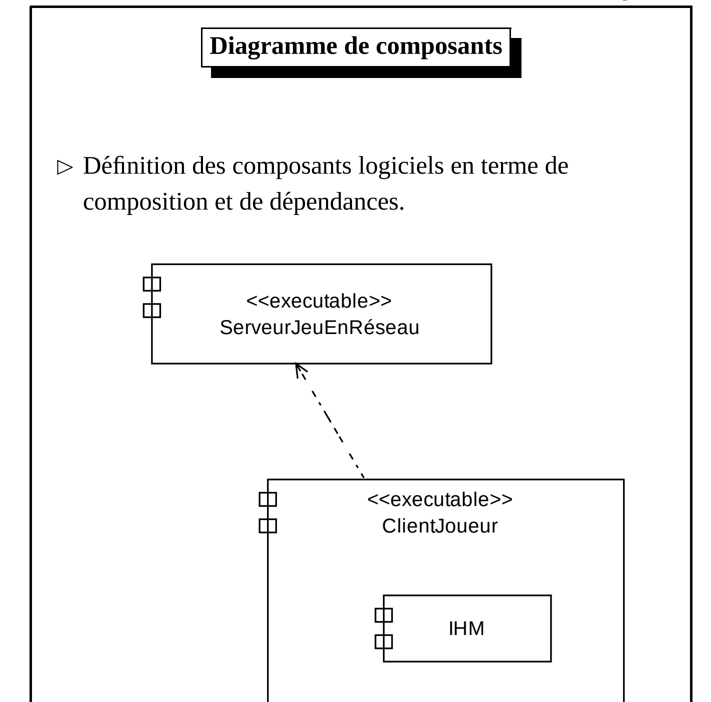
▷ Stéréotypes prédéfinis :
- « exécutable » ;
- « library » ;
- « file » ;
- « document » ;
- « table » ;
- « EntityBean » et « SessionBean »
(support pour les EJB en cours de standardisation).

▷ Liens possibles entre composants : *dépendance* et *composition*.

▷ Comme une classe, un composant peut « réaliser » une (ou plusieurs) *interface(s)*.

Diagramme de composants

▷ Définition des composants logiciels en terme de composition et de dépendances.



Noeud

▷ Ressource matérielle utilisée lors de l'exécution
⇝ exemples : PC, serveur Web, Imprimante, ...
(pas de stéréotypes prédéfinis).

▷ Distinction classe vs instance
⇝ exemple : PC vs <u>monPC : PC</u>

▷ Les noeuds peuvent accueillir des *composants*
(ou plus précisément des instances de composants).

▷ Les noeuds sont reliés par des *liaisons*
⇝ exemples : Ethernet, Internet, ...
⇝ possibilité de spécifier le débit.

Diagramme de déploiement (1)

▷ Définition des noeuds en tant@qu'unités physiques d'exécution ;

▷ Définition des liaisons en tant que liens de communication.

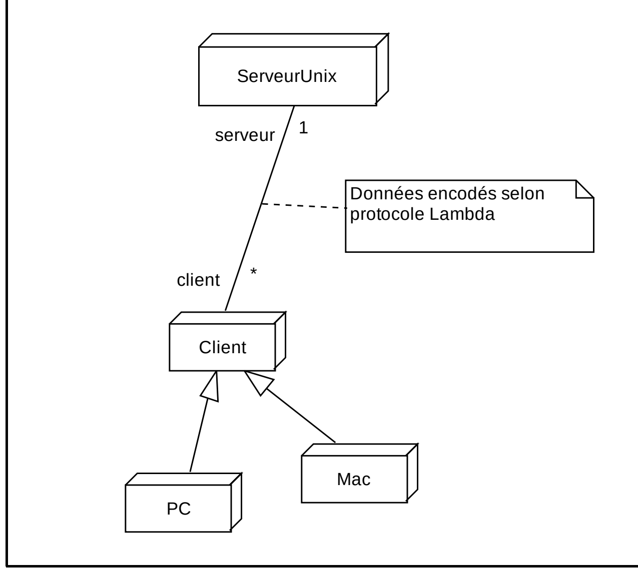
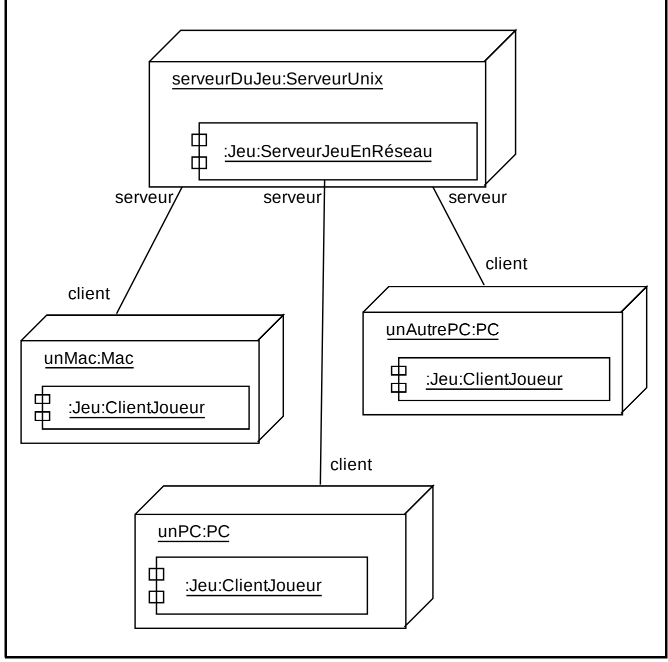


Diagramme de déploiement (2)

▷ Définition de la répartition des composants sur les différents noeuds.



Directives (ou « tagged values »)

Mécanisme général permettant d’annoter le modèle en vue (principalement) de son implantation.

▷ Tout élément du modèle peut être annoté : package, classe, opération, paramètre, association, etc.

▷ Sous Modelio : directives interprétées par des règles d’*hypergénericité*.

▷ Exemples de directives pré-définies pour les opérations :
- *virtual* et *inline* (C++ seulement), *JavaNative*, ...

▷ Exemples de directives pré-définies pour les attributs :
- *JavaShort*, *JavaLong*, *JavaGenerateAccessor*, ...
- *, & : génération d’un pointeur vs référence (C++)
- *short*, *long*, *const*, *create*, *fullaccess*, *array*, ... (C++)

▷ Exemples de directives pré-définies pour associations :
- *JavaGenerateAccessor*, *ordered*, ... (Java)
- *access fullaccess*, *ordered*, ... (C++)

Hypergénéricité

L’hypergénéricité est un mécanisme de Modelio à base de règles (langage J) permettant :

▷ de rendre paramétrable la génération de code (et de documentation) ;

▷ d’automatiser au maximum la phase de conception technique et de codage.

Principe :

▷ on utilise les directives prédéfinies sous Modelio pour annoter les éléments de modélisation lors de la phase de conception ;

▷ si besoin, on peut définir de nouvelles directives et modifier le jeu de règles pilotant la génération de code ;

▷ l’idée est de spécifier des principes à un niveau global, quitte à les invalider localement ;

▷ le code généré tient alors compte des directives choisies.

Méthodologie de la conception à l’intégration

1. **Conception préliminaire et intégration**.
2. **Conception détaillée et tests unitaires**.
3. **Réalisation : d’UML à C++ / Java**.
4. **Document de conception**.

Conception préliminaire (ou « system design »)

Huit étapes (pas forcément séquentielles) :

C1. Identifier les objectifs de conception.

C2. Décomposer le système en sous-systèmes.

C3. Ré-utiliser (frameworks et bibliothèques de classes).

C4. Définir le déploiement des composants sur les noeuds.

C5. Décider le mode de stockage des données.

C6. Décider une politique de contrôle d'accès.

C7. Décider la nature du flux de contrôle global.

C8. Identifier les conditions aux limites.

... avant passage à la conception détaillée (ou « object design »).

C1 : identifier les objectifs de conception

▷ Expliciter et ordonner les objectifs de conception parmi :

- réduction des **coûts** de développement et/ou d'installation et/ou de fonctionnement ;
- réduction des **délais** de livraison du prototype et/ou du système complet ;
- maximiser les **performances** : temps de réponse, nombre de transactions, besoin en mémoire, etc. ;
- maximiser la **fiabilité** : robustesse, disponibilité, sécurité, etc.
- maximiser l'**extensibilité**, la **portabilité**, l'**ergonomie**, etc.

▷ Attention : ces objectifs sont souvent incompatibles :
⇝ temps de réponse *vs* besoin en mémoire.
⇝ rapidité *vs* taille de l'exécutable.
⇝ rapidité de réalisation *vs* autres critères.

▷ On pourra s'appuyer sur les besoins non-fonctionnels identifiés au début de l'analyse.

C2 : décomposer le système en sous-systèmes

▷ S'appuyer sur la décomposition en packages issue de l'analyse.

▷ Principes :

- regrouper par fonctionnalité ;
- minimiser le nombre de dépendances entre sous-systèmes ;
- tenir compte de l'architecture du système (monolithique, distribuée, ... voir C4) ;
- tenir compte des objectifs de conception, notamment en termes de performances.

▷ Utiliser les diagrammes de composants pour présenter la décomposition en sous-systèmes.

C3 : ré-utiliser (frameworks et bibliothèques de classes)

▷ Il s'agit :

1. d'identifier les classes qu'on *pourrait* éviter de réaliser « from scratch » ;
2. pour ces classes, d'évaluer la pertinence d'une réalisation « from scratch » selon des critères à définir parmi :
   - fiabilité, performance, portabilité, ...
   - minimisation des risques, des coûts et/ou délais.

▷ Remarque : utiliser un framework peut conduire à des choix imposés (en termes de flux de contrôle notamment, voir C7).

▷ Bien comprendre les différences entre :

- **bibliothèques de classes** ;
- **frameworks** ;
- **patterns**.

A propos de « design patterns »

▷ Synonymes : « patron » ou « schéma de conception ».

▷ Origine : ouvrages de l'architecte Christopher Alexander (The Timeless Way of Building, A Pattern Language)

▷ Objectifs :
- ◦ identifier/nommer/diffuser les meilleures pratiques ;
- ◦ accélérer la communication et les prises de décisions durant les phases d'analyse et (surtout) de conception.

▷ Comporte *systématiquement* :
– titre, diagramme, propos, description, exemple(s),
– et si besoin : contre-indications, patterns connexes, etc.

▷ Exemples de patterns : composite, abstract factory, adapter, MVC, etc.

▷ Cf. exemples plus loin dans le cours... et Bibliographie.

C4 : définir le déploiement des composants

▷ Valider un choix d'architecture :
- ◦ nombre, nature et caractéristiques des noeuds ;
- ◦ nature et caractéristiques des liaisons ;
- ◦ le cas échéant, OS, SGBD, etc.

▷ Définir le déploiement des composants sur les noeuds :
- ◦ traiter les problèmes induits par le déploiement : transferts des données, synchronisation, sécurité, etc. ;
- ◦ se souvenir que le logiciel survit généralement au matériel $\Longrightarrow$ minimiser l'impact d'un portage sur une autre architecture.

▷ Utiliser les diagrammes de déploiement pour présenter l'architecture (noeuds et liaisons) et le déploiement des composants sur les noeuds.

C5 : décider le mode de stockage des données

▷ Il s'agit de maîtriser la gestion des données stockées en mémoire *permanente* (sur disque) :
- ◦ données à stocker entre 2 exécutions du programme ;
- ◦ données trop volumineuses pour la mémoire vive ;
- ◦ données à partager avec d'autres programmes ;
- ◦ données « sensibles » qu'il faut être sûr de préserver même en cas de crash.

▷ Identifier les données à stocker (objets « persistants »).

▷ Décider mode de stockage parmi : simple fichier, BDD relationnelle, BDD objet.

▷ Critères :
- ◦ taille et nature des données à stocker ;
- ◦ complexité des requêtes ;
- ◦ performance souhaitée (temps d'accès et mémoire).

C6 : décider une politique de contrôle d'accès

▷ Il s'agit de gérer les différences d'accès (éventuelles) entre les différents acteurs :
- ◦ accès plus ou moins étendu aux fonctionnalités du système ;
- ◦ accès aux données créées par l'utilisateur lui-même ;
- ◦ accès aux données créées par les autres utilisateurs ;
- ◦ accès aux données internes au système ;
- ◦ accès aux données « confidentielles » ou « stratégiques ».

▷ Identifier les données et fonctionnalités accessibles à chaque acteur.

▷ Mettre en place les mécanismes pour gérer les différents niveaux d'accès au système (fonctionnalités et données).

▷ Mettre en place les mécanismes pour authentifier les utilisateurs.

C7 : décider la nature du flux de contrôle global

▷ Il s'agit, *pour chaque composant qui s'exécute sur un processus propre*, de choisir parmi :

- le mode **procédural** : le programme enchaîne les actions selon un ordre strictement déterminé en s'interrompant seulement lorsqu'il attend des données de l'utilisateur ou d'un autre programme...
  - ⇝ adapté aux systèmes à l'IHM en mode texte ;
  - ⇝ typique des langages procéduraux.
- le mode **réactif** : une boucle globale traite les événements selon leur arrivée et les transmet aux objet(s) concerné(s)...
  - ⇝ adapté aux systèmes à l'IHM relativement simple.
- les **threads** : le programme se compose de processus « légers » qui s'exécutent en // et réagissent à des événements différents...
  - ⇝ adapté au systèmes complexes ;
  - ⇝ élégant, intuitif, mais délicat à mettre au point.

C8 : identifier les conditions aux limites

▷ Il s'agit de préciser :

- les phases de **démarrage** et d'**arrêt** du système ;
- le cas échéant, l'**interruption** du système ;
- le cas échéant, les modes de **fonctionnement « dégradés »** (panne de disque dur, de courant, de réseau, ... ou simple maintenance).

▷ Les aspects à prendre en compte incluent :

- vitesse de démarrage ;
- intégrité des données en toutes circonstances ;
- bloquages d'accès sélectifs (maintenance) ;
- gêne occasionnée aux *autres* systèmes lorsque le logiciel est arrêté, interrompu, en cours de démarrage, ... ;
- information des différents acteurs lorsque le système n'est pas dans un état de fonctionnement normal.

Conception préliminaire : résumé

La conception préliminaire consiste en :

1. un certains nombre de *choix stratégiques globaux* :
   - ▷ stockage des données et contrôle des droits d'accès ;
   - ▷ frameworks et classes pré-existantes sélectionnées ;
   - ▷ nature du flux de contrôle ;
   - ▷ organisation du système en sous-systèmes ;
   - ▷ choix de l'architecture en terme de noeuds et liaisons.
2. un certains nombre d'*ajouts* et de *modifications* dans les modèles issus de l'analyse :
   - ▷ ajouts des diagrammes de composants et de déploiement ;
   - ▷ modifications des autres diagrammes pour prendre en compte les choix ci-dessus ainsi que les conditions aux limites ;
   - ▷ précisions documentaires relatant les choix de conception et les raisons de ces choix (traçabilité).

Conception préliminaire : bilan

A la fin de cette étape :

▷ toutes les décisions globales (stratégiques) ont été prises et documentées ;

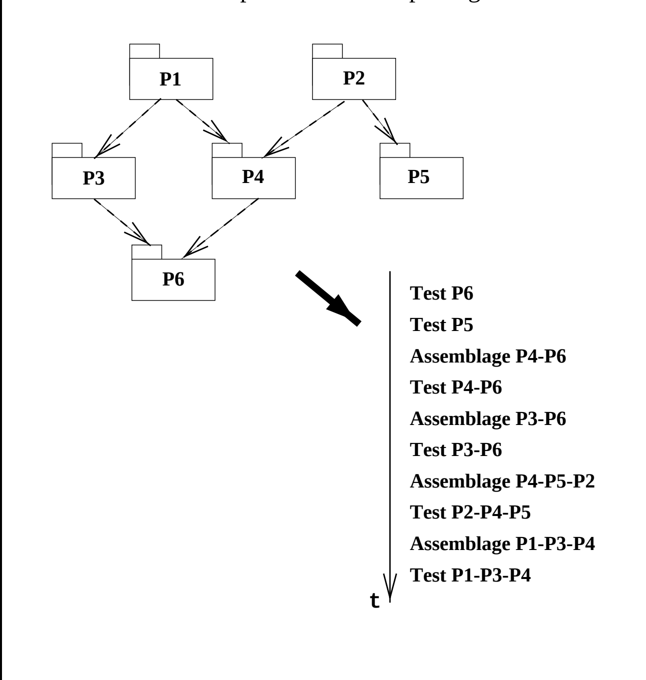
▷ tous les packages sont identifiés et chaque classe appartient à un package et un seul ;

▷ il sera donc possible de paralléliser la suite du développement en confiant chaque package à une équipe ;

▷ il est possible, et souhaitable, de préparer sans attendre l'intégration...

Intégration : ordonnancement des travaux

▷ Se baser sur les dépendances entre packages :



Conception détaillée (ou « object design » ou « class design »)

Quatre étapes (pas forcément séquentielles) :

C9. Compléter la spécification.

C10. Frameworks et bibliothèques de classes.

C11. Restructurer.

C12. Optimiser.

... avant passage à la réalisation.

Objectif : rapprocher, par étapes successives, la modélisation issue de l'analyse préliminaire de sa concrétisation informatique à venir.

C9 : compléter la spécification

Il s'agit de terminer la description de tous les éléments à implanter :

▷ ré-examen des classes, notamment les classes techniques ajoutées lors de la conception ;

▷ finaliser les **APIs** (interfaces de programmation)
⇝ travailler au niveau des **signatures** en vérifiant la cohérence (ordre des paramètres, nommage, ...) ;

▷ expliciter les **pré-/post-conditions** de chaque méthode ;

▷ préciser les **exceptions** ;

▷ préciser les **visibilités** ;

▷ ajouter les **directives** d'implantation ;

▷ éventuellement, décrire le corps de chaque méthode en utilisant les méthodes d'accès et de manipulation des relations et des attributs (voir plus loin).

C10 : Frameworks et classes ré-utilisés

▷ Il s'agit de :

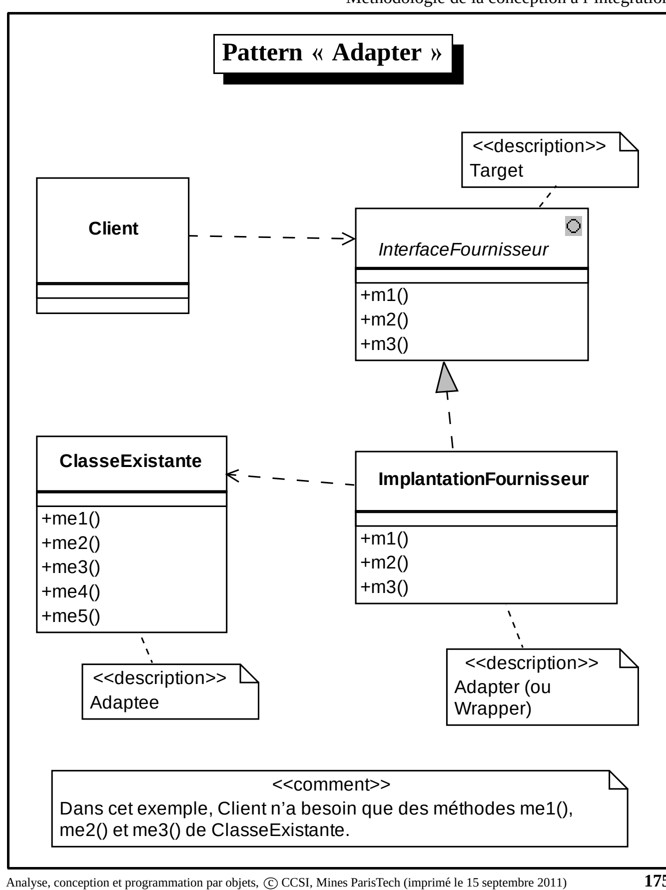
- paramétrer les frameworks et les classes ré-utilisées ;
- modifier la modélisation pour tenir compte des frameworks et bibliothèques de classes utilisées.

▷ Aussi, il est prudent :

- d'anticiper l'évolution des frameworks et classes ré-utilisées ;
- de prévoir leur remplacement éventuel par d'autres plus performants (ou moins coûteux).

⇝ On tachera donc de découpler au maximum ces classes des autres classes du système.

Pattern « Adapter »



Exercice : le pattern « Singleton »

Comment faire en sorte qu'une classe ne puisse être instanciée qu'une fois au maximum ?

C11 : Restructurer

Il s'agit, *pour chaque diagramme de classes,* de traiter les cas suivants :

▷ suppression des généralisations multiples si le langage cible ne les permet pas ;

▷ transformation des associations n-aires en associations binaires ;

▷ remplacement des classes d'association par des classes ordinaires ;

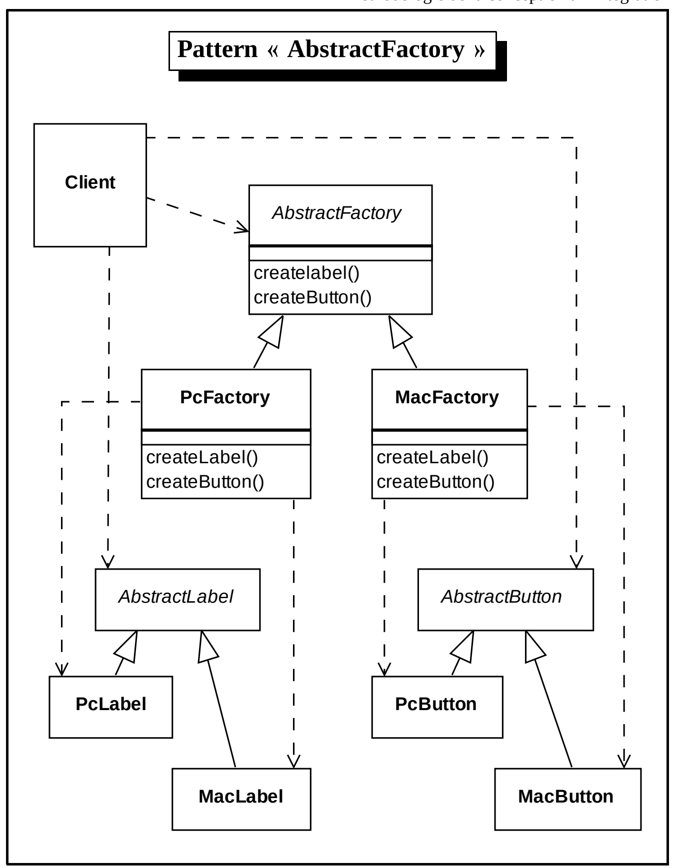
▷ ... le tout en maintenant dans la mesure du possible la *lisibilité* des diagrammes.

En fonction des objectifs, il peut s'agir aussi de :

▷ préparer la réutilisation de certaines classes ;

▷ augmenter la portabilité du système
(vis-à-vis des classes techniques d'IHM par exemple).

Pattern « AbstractFactory »



Pattern « AbstractFactory » (utilisation)

```
AbstractFactory fac = newPcFactory();

// La suite du code est portable (ne
// dépend pas de la plate-forme utilisée)

AbstractButton button=fac.createButton();
AbstractButton button2=fac.createButton();

AbstractLabel lab=fac.createLabel();

...
```


C12 : Optimiser

Les techniques suivantes devront être utilisées à bon escient :

- ▷ ajout de raccourcis dans les chemins entre classes (pour l'accès aux opérations les plus fréquemment invoquées) ;
- ▷ accélération des accès au niveau des associations de cardinalité élevée ou variable (via *ordonnancement* des instances ou liaison directe vers *certaines* instances) ;
- ▷ déplacement d'attributs vers la classe qui les utilise directement ;
- ▷ suppression des classes « inutiles » (via rapatriement des données/méthodes dans la classe utilisatrice) ;
- ▷ ajout de dispositifs pour mémoriser les résultats des opérations coûteuses en temps (« cache »), pour « bufferiser » certains accès coûteux, etc.

Conception détaillée : résumé

La conception détaillée consiste en :

- ▷ un certains nombre de *choix techniques* délicats :
  - ○ paramétrage et encapsulation des composants/classes/frameworks ré-utilisés ;
  - ○ nature des optimisations à effectuer.
- ▷ un certains nombre d'*ajouts et modifications* dans les modèles issus de la conception préliminaire :
  - ○ restructurations des différents diagrammes de classes pour, notamment, prendre en compte les choix techniques ci-dessus ;
  - ○ spécification finale des APIs, contraintes, pré-/post-conditions, exceptions, ...
  - ○ visibilités ;
  - ○ ajouts de précisions documentaires, notamment : descriptions techniques des méthodes, ...

Conception détaillée : bilan

A la fin de cette étape :

- ▷ toutes les décisions de détail ont été prises et documentées ;
- ▷ chaque entité est complètement spécifiée ;
- ▷ il est donc temps de passer à la réalisation de chaque package ;
- ▷ il est possible, et souhaitable, de préparer les tests unitaires...

Réalisation : d'UML à C++

UML	⇝	C++
classe	⇝	class
attribut publique	⇝	attribut protégé avec méthodes publiques d'accès
attribut protégé	⇝	attribut protégé avec méthodes d'accès protégés
attribut privé	⇝	idem mais tout « privé »
opération	⇝	fonction membre
généralisation	⇝	héritage (public par défaut)
association	⇝	collection (liste, tableau, ...) avec méthodes d'accès
package	⇝	namespace
invariants, pré- et post-conditions	⇝	assert($condition$)

Réalisation : accès aux associations depuis C++

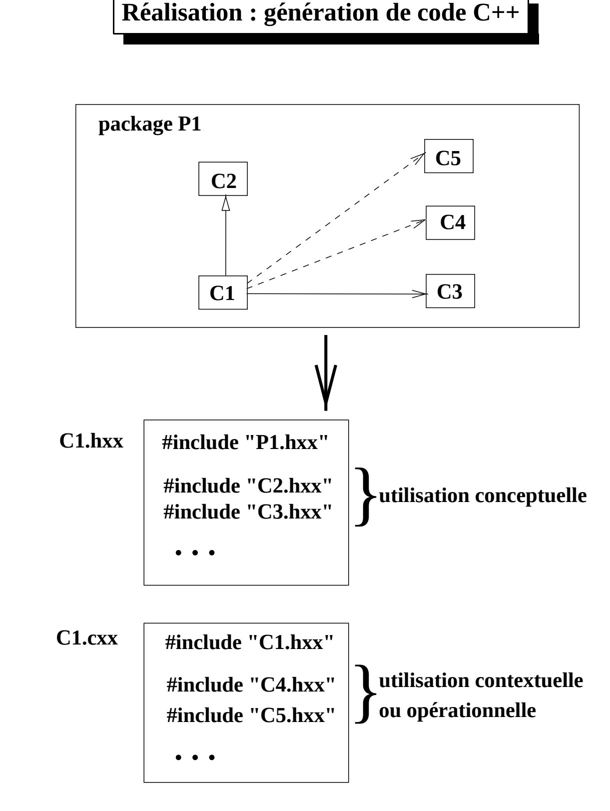
Pour chaque association, 4 méthodes sont automatiquement ajoutées à la classe utilisatrice (cela dépend toutefois de la configuration du projet). Soit, par exemple, une classe C1 utilisant une classe C2 avec comme rôle r, on a les 4 méthodes suivantes définies dans C1 :

- `int card_r()`
- `C2* get_r(int i=0)`
- `void append_r(const C2*)`
- `void erase_r(const C2*)`

Autres possibilités :
- gérer les associations à la main en annotant l'association par la directive (ou «tagged value») `own` ;
- cas particuliers : composition, classe d'association, association « de classe ».

⇝ Cf. la doc Modelio pour plus de détails sur la traduction en C++ des concepts UML.

Réalisation : génération de code C++



Réalisation : d'UML à Java

UML	⇝	Java
classe	⇝	classe publique
attribut publique	⇝	attribut protégé avec méthodes publiques d'accès
attribut protégé	⇝	attribut protégé avec méthodes d'accès protégés
attribut privé	⇝	idem mais tout « privé »
opération	⇝	fonction membre
généralisation	⇝	héritage
association	⇝	attribut de type ArrayList + méthodes d'accès
package	⇝	paquetage
invariants pré- et post-conditions	⇝	`if (!condition) throw new RuntimeException`

Réalisation : accès aux associations depuis Java

Pour chaque association, 6 méthodes sont automatiquement ajoutées à la classe utilisatrice (cela dépend toutefois de la configuration du projet). Soit, par exemple, une classe C1 utilisant une classe C2 avec comme rôle R, on a les 6 méthodes suivantes définies dans C1 :

- `int cardR()`
- `C2 getR(int)`
- `void setR(int,C2)`
- `void appendR(C2)`
- `void eraseR(int)`
- `void eraseR(C2)`

Cas particuliers : composition, classe d'association, association « de classe ».

⇝ Cf. la doc Modelio pour plus de détails sur la traduction en Java des concepts UML.

Réalisation : génération de code Java

C1.java

```
package P1;

import P2.*;

public class C1 extends C2 {
   protected ArrayList<C3> R = new ArrayList<C3>();
   ...
};
```


Document de conception : plan type

1. Présentation générale
   ▷ Situation et objectifs, documents de référence, etc.
2. Architecture
   ▷ Contraintes d'implantation, architecture technique, principes d'implantation.
3. Conception générale
   ▷ Définition complète du modèle, package par package :
   - package P_1 :
     classes C_1, C_2, ... C_n
     chacune accompagnée le cas échéant de ses :
       invariants, pré-/post-conditions, diagrammes d'états, ...
   - package P_2 :
     ...
4. Traçabilité
5. Plan d'intégration

Tests

1. **Définitions**.
2. **Tests de fiabilité**.
3. **Tests fonctionnels**.
4. **Tests non-fonctionnels**.
5. **Profiling et optimisation**.
6. **Tests de portabilité**.
7. **Quand s'arrêter de tester ?**.

Définitions

▷ Il faut comprendre « les **tests** » comme la recherche *active* et *systématique* de différences entre le fonctionnement supposé du programme et son fonctionnement effectif (en termes de fonctionnalités, de fiabilité et de performances).

▷ On appelle **cas de test** (ou **test case**), un ensemble composé des éléments ci-dessous

1. *nom* du cas de test ;
2. *input* (données et/ou commandes) ;
3. *oracle* (résultat attendu) ;

... et destiné à mettre en évidence une faute dans une partie du programme.

Tests de fiabilité

Analyse dynamique du code lors de l'exécution :

▷ détection d'incohérences dans les classes entités, de non-respect du mode d'emploi des méthodes, ...

⇝ Test systématique des pré-/post-conditions et invariants.

Nota-Bene : il vaut mieux prévenir que guérir.

▷ détection de problèmes mémoire (surtout utile pour langages sans garbage collector comme C ou C++) ;

▷ détection des parties de code jamais parcourues à l'exécution lors des tests
⇒ code suspect : inutile et/ou jamais testé !

⇝ Utilisation d'outils tels que Rational PurifyPlus (IBM).

▷ Relecture systématique du code (cf. méthodologie XP).

Tests fonctionnels

Impossible de *tout* tester (surtout dans le cas d'une IHM graphique), par contre on peut :

▷ Tester un maximum de points
⇒ prévoir du temps pour les tests ;

▷ Faire tester par un maximum de personnes différentes (informaticiens *et* non-informaticiens) ;

▷ Accumuler les cas de tests
⇝ batterie de centaines (voire milliers) de cas de tests ;

▷ Automatiser l'application et l'analyse des cas de tests
⇒ utiliser des outils adaptés (y compris des « robots » pour tester les IHMs).

Batterie de tests à ré-appliquer systématiquement sur chaque nouvelle version du code
⇝ **tests de non-régression**.

Tests non-fonctionnels

▷ Vérifier les *performances* du logiciel :
- temps de calcul ;
- besoin mémoire à l'exécution ;
- en cas d'IHM graphique : réactivité de l'IHM, fluidité des animations, ...
- etc.

▷ Cas particulier des « tests de charge » :
- de plus en plus nécessaires (Web, e-commerce, ...) ;
- nécessite un outil ad hoc pour *simuler* des milliers (voire des millions) de requêtes pendant un temps défini ;
- résultats souvent imprévisibles avant le test
⇒ tester sur un prototype le plus tôt possible.

Profiling et optimisation

Si besoin (en cas de performances insuffisantes) . . .

▷ Etude dynamique du code à l'exécution : mise en évidence des parties du code à optimiser en priorité (où passe-t-on du temps à l'exécution ?)
⇝ utilisation d'outils tels que prof (Unix) ou Rational PurifyPlus (IBM).

Tests de portabilité

▷ Compilation et tests (fonctionnels et performance) de chaque exécutable sur les différentes architectures visées (matériels × OSs × autres logiciels).

▷ Si nécessaire, tester avec des périphériques d'E/S différents :

- écran basse vs haute résolution ;
- écran couleur vs monochrome ;
- claviers de type différents ;
- ...

Exercice : quand s'arrêter de tester ?

Les tests permettent de déceler des erreurs, jamais de garantir qu'il ne reste plus d'erreurs dans le code...

⇝ Comment évaluer le nombre N d'erreurs non encore décelées dans le programme ?

⇝ Est-il possible d'avoir une idée a priori de la fiabilité du logiciel produit ?

Bibliographie

1. **Ouvrages de référence**
2. **Quelques autres ouvrages sur l'Objet, UML, les patterns, etc**
3. **Quelques URLs**

Ouvrages de référence

... sur le langage de modélisation UML : [VF : Eyrolles]

– Grady Booch, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 2005, 500p.
– James Rumbaugh, Ivar Jacobson, Grady Booch. *UML 2.0 Guide de référence*. Campus Press, 2004, 774p.
– Jos Warmer, Anneke Kleppe. *The Object Constraint Language : Precise Modeling with UML (2nd ed.)*. Addison-Wesley, 2003, 232p.

... sur les méthodologies « Unified process » :

– Ivar Jacobson, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999, 463p. [VF : Eyrolles]
– Jim Arlow, Ila Neustadt. *UML 2 and the Unified Process (2nd ed.)*. Addison-Wesley, 2005, 592p.

...pour découvrir les méthodologies « légères » :

– Kent Beck. *eXtreme Programming*. Campus Press, 2002, 230p.
– Alistair Cockburn. *Crystal Clear - A Human-Powered Methodology for Small Teams*. Addison-Wesley, 2004, 312p.
– Henrik Kniberg. *Scrum et XP depuis les Tranchées*. C4Media Inc, 2007, 150p. Disponible gratuitement en-ligne (www.infoq.com/minibooks/scrum-xp-from-the-trenches)

Quelques ouvrages sur l'Objet, UML, les patterns, etc.

... pour approfondir UML (et les patterns) :

– Bernd Bruegge, Allen H. Dutoit. *Object-Oriented Software Engineering : Using Uml, Patterns and Java (2nd edition)*. Prentice-Hall, 2003, 800p.
– Michael R Blaha, James Rumbaugh. *Modélisation et conception orientées objet avec UML 2* Pearson Education, 2005, 585p.

...pour approfondir les « patterns » :

– Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns*. Addison-Wesley, 1995, 395p. [VF : Eyrolles]
– Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates. *Design patterns tête la première* O'Reilly, 2005, 639p.

... pour approfondir gestion de projets & génie logiciel :

– Shari L, Pfleeger. *Software Engineering, Theory and Practice (2nd Edition)*. Prentice-Hall, 2001, 659p.
– Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 2003, 604p.

Quelques URLs

`http://www.omg.org`
⤳ Site de l'OMG
⤳ Documentation sur UML (normalisé par l'OMG)

`http://www.modeliosoft.com`
⤳ Site sur l'outil Modelio (voir aussi `www.softeam.fr`)

`http://www.rational.com`
⤳ Site de la division d'IBM qui commercialise les outils Rational, le RUP, ...
⤳ UML Resource Center

`http://www.extremeprogramming.org/`
`http://www.xprogramming.com/`
⤳ Sites dédiés à l'Extreme Programming (XP)

`http://www.agilemodeling.com/`
⤳ Site de référence pour l'Agile Modeling (AM)